Eindhoven, July 28, 2008

Design and verification of the SmartPixel II protocol

by
Rink P. W. Springer

in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science and Engineering

Supervisor:
M. A. Reniers, TU/e department of Mathematics and Computer Science
W. Hendriksen, Devlab

Preface

As systems keep on growing more and more complicated, the need to formally analyze such systems becomes evident: pure intuition and experience are no longer sufficient. An interesting way to analyze the behavior of systems is by means of process theory: a system can be described in terms of processes. Such a specification can be used to subsequently prove desired properties, either by model inspection or by using temporal logic. This thesis details the process of transforming an external specification to a formal process specification and validating whether desired properties hold using the mCRL2 toolkit. The latter is especially of interest as the analyzed system heavily depends on the notion of time, which has prompted interesting challenges during the modeling and verification process.

Acknowledgments

First and foremost, this thesis would not be its current form without the help of my supervisor, Michel Reniers. We regularly discussed the current results and reflected upon the status of the work, which both have really aided me during this project. I would also like to thank Simon Janssen for the useful discussions and constructive criticism while performing the project, as well as all other persons who have read my thesis and provided useful comments. Last but not least, the mCRL2 development team has been very responsive in regard to my bug reports - I would especially like to thank Jeroen van der Wulp for his patience and help.

Eindhoven, July 2008
Rink Springer

Abstract

The SmartPixel system consists of a set of independent nodes, which have to cooperate in order to accomplish desired functionality. This makes the system a good example of a completely distributed system: all nodes have to work together to achieve the intended result. The result is that the SmartPixel system is very well suited for a detailed analysis.

In the thesis, we will first present the SmartPixel system and provide an introduction to the process theory used. From then on, the different phases of the system will be individually modeled and validated, after which the focus will be on the system as a whole. The remainder of the thesis will attempt to transform timed models to untimed models in an attempt to prove more properties of the system. Finally, the findings are summarized in the concluding words.

Contents

Chapter 1

Introduction

DevLab is an organization performing scientific and applied research, where the goal is to provide extra applicable knowledge to participating companies. All of the thirteen participating companies are small- to medium size companies active within the technology sector, and together they are part of Development Club, which in turn is part of FHI, an institution dedicated to industrial electronics. DevLab cooperates with several universities to allow a number of PhD students to perform academic research. These students are in turn assisted by academic and college students in order to obtain knowledge that allows the participating companies to create new products and services.

There are a number of projects being conducted within the DevLab organization, one of which is Dutch Clay: an intelligent form of clay consisting of small balls or cells. There are almost limitless applications imaginable; think of three dimensional image storage: a bag filled with Dutch Clay, in which you insert your arm and instruct the clay to remember their configuration. Subsequently removing your arm from the bag, shaking the bag a few times in order to mix the clay and instructing the clay to restore the configuration would result in the clay displaying the exact same arm you previously inserted. Or what about using the clay as a distributed system, which implies adding more clay would increase the processing power.

One of the main challenges in the Dutch Clay system is determining the position of the clay. This has been researched in great detail and this problem currently seems unfeasible in an energy and space-constrained application such as this one. To this end, the clay cells were replaced with cubes in an attempt to simply determining the position of the clay. It turned out that this simplification was not enough to make the system feasible, so subsequently, another simplification was introduced: the problem was reduced to 2D by only considering squares, which are called *SmartPixels*, each connected with up to four adjacent SmartPixels. A network of these SmartPixels can then display predefined figures. The complete overview of the desired functionality is described in Section 2.

A current sketch of the system has been provided in [Hen08]. This sketch discusses a number of states and conditions which need to hold to perform state transitions. Each state performs a series of actions, which are described in an algorithm. However, these descriptions are very superficial and based on pure intuition: no attempts have been made to verify whether a system based on this sketch will behave as intended. This is the overall goal of the assignment: use formal modeling methods to describe such a system, in order to determine whether the algorithms described in the sketch work as intended. A second goal is to determine if and how this system can be improved: once a model of the system has been constructed, it can be used to gain more insight in the operation of the system. Using such insights, there may be possible improvements. Finally, the thesis should describe the model in such detail that readers are able to update the model if they alter the system to see if desired properties still hold. A concrete list of research questions is

presented in Section 4.6.

Chapter 2

Overview

A single *SmartPixel* is a square device, which contains a light that can be switched on or off. The device also contains four physical connectors, referred to as $0, \ldots, 3$, which reside on each side and are used in order to communicate with a neighboring SmartPixel. A number of these SmartPixels connected together form a network. We shall refer to a single SmartPixel in such a network as a *node*. It is important to note that all nodes contain completely identical hardware and software; the only property that makes them distinguishable is an *identification number*, or *id*, which is unique per node. Since the nodes are square-sized, they can be rotated any multiple of 90° before they are connected to adjacent nodes. Nodes know nothing of their position in the network, let alone whether each side has a neighbor - all they can do is send messages to adjacent neighboring nodes, receive messages from such nodes and switch the light on or off. A visual representation of such a SmartPixel can be found in Figure 2.1.

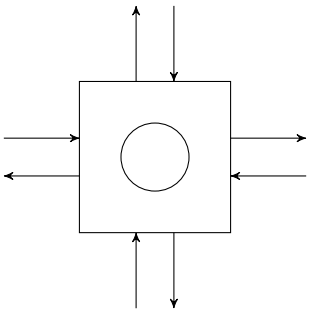


Figure 2.1: Visual representation of a single SmartPixel

Once such a network of nodes has been assembled, connected and switched on, the nodes will perform some necessary initialization steps in order to determine the network configuration; these steps are outlined in Section 4. Once the initialization is completed, the nodes continue to perform their main function: display the first figure out of a list of figures, delay a few seconds, display the next figure, delay a few seconds, etc. This keeps going until we pull the plug. If nodes are physically altered, such as by adding or removing nodes while the system is operating, the system should adapt to the new situation. For the purposes of these figures, it is assumed that the network itself is square-sized, $N \times N$ nodes. A possible configuration is presented in Figure 2.2: each square is a node, where the 0, 1, 2, 3 denote the north, east, south and west connections of this node. Finally, the number represents the unique identification number of the node.

While the usefulness of such a system can be doubted, the interesting challenges that will arise while specifying and proving a model cannot. How will nodes know 'what to do'? How can we

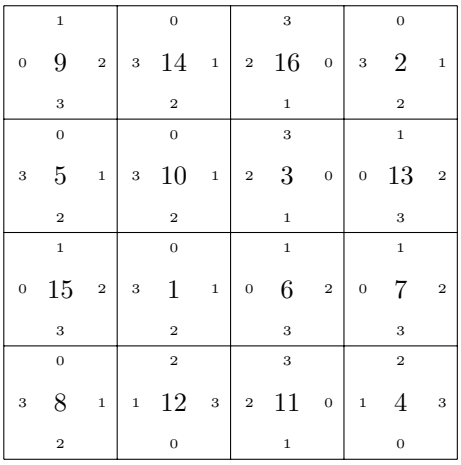


Figure 2.2: Example of a 4×4 network of SmartPixel nodes

ensure that the system remains as operational as possible whenever some nodes fail? Will the algorithms proposed in Section 4 turn out to be correct, or will unforeseen side-effects show up in the model?

2.1 Hardware

The hardware in use consists of a PCB board of 3×3 cm, containing an ATMega644 microcontroller and some assorted parts. A message can be sent out to each of the four directions individually and it is possible to tell from which direction a message was received. In each direction, only a single message can be sent at a time; these messages are transmitted using Manchester encoding. While a message is being sent, an incoming message will still be received.

2.2 Assumptions

Considering the hardware and requirements of the system, this design is based on the following assumptions:

1. Each node may have a slight clock speed deviation
   There may be a slight deviation ($\pm 10\%$) from the specified clock speed - this deviation is unknown to the node.
2. Nodes can be added, removed and switched on/off at any time in the system.
3. The delay between node A transmitting a message and node B receiving it is negligible.
4. It is possible to distinguish between a valid and corrupt message.
   We will only consider retrieving valid messages; corrupt messages will always be discarded.

Chapter 3

Introduction to process algebra

3.1 Basics of process algebras

The foundation of the formal modeling methodology as used throughout the thesis is known as *process algebra*: a process algebra is an abstract method to describe the behavior of a system from a high-level perspective. The process algebra used in the thesis is the one used by mCRL2 [GMR$^+$07] as described in [GR07], which in turn is based on ACP [BK84]. Process algebras are specified using axioms; axioms can be seen as rules which are always valid. All axioms are presented in Appendix B.

Generally, a system is divided into a number of *processes*. The idea of this division is that processes are independent: they only influence each other when explicitly modeled that they do, by means of communication (this will be discussed in Section 3.3). Of course, these processes need to work together to hopefully achieve the desired effect. We refer to the set of all processes as the *system* we are modeling. For example, if we were to model a coffee vending machine, there may be a process $\mathcal{U}$ responsible for the user input, a process $\mathcal{P}$ responsible for handling payment and a process $\mathcal{S}$ responsible for serving the coffee. If money has been inserted, the payment process should inform the user interface, to allow the user to see how much money has been inserted into the machine. Likewise, if a flavor has been selected and enough money has been inserted into the machine, the serving process should be instructed to start brewing coffee.

In the example above, the overall coffee vending system consists of processes $\mathcal{U}$, $\mathcal{P}$ and $\mathcal{S}$, which all run in parallel. As these processes run in parallel, it is possible to combine them into a single process yielding the same behavior as all processes in parallel would, by means of the *parallel operator* $\|$. The result is that we can obtain a process $\mathcal{M}$ illustrating the behavior of the entire coffee vending machine, for which it holds that $\mathcal{M} = \mathcal{U} \parallel \mathcal{P} \parallel \mathcal{S}$. This operator will be covered in Section 3.7 - for now, the most important notion is the existence of the operator.

3.2 Actions

Modeling can generally be regarded as describing a system from a high-level perspective: most implementation details, for example the precise steps required to communicate a message from one system to another are not discussed: such implementation details are rarely interesting from such a high-level abstraction. This brings us to the notion of *actions*: basically, an action is an atomic event which may or may not occur - for example, we may have an action *opendoor*, which illustrates that some door has been opened and an action *closedoor* which illustrated the same door has been closed. It must be noted that a system designer is always free to chose the action

names; claiming action a illustrates that some door has been opened is fine as well. To this end, an overview mapping each action to an atomic event should always be included in every model to prevent possibly confusion or ambiguity. For the remainder of this chapter, we shall use action names a, b, c, etc. if we need to refer to some arbitrary action without a special function.

One may wonder, since we appear to only support actions, how events are being represented within our formalism. This introduces some food for thought: what is the difference between an action and an event? A natural response is that an action represents a change initiated by the process, whereas an event represents a change initiated by the environment. Yet, this distinction is actually confusing, since if we were to create a model detailing the interactions between two processes, a change initiated by the first process would be an environment change for the second process, and the other way around! From this, it follows that distinguishing between actions and events is not a very good idea: it makes matters much more confusing.

Based on the reasoning above, any atomic event can be encoded as an action, like opening/closing doors, but there could be an event *isDoorOpen*, which indicates a door is sufficiently opened to allow a person to move through it. If we want to check whether the door is sufficiently opened, we attempt to issue a *isDoorOpen* action. If this is possible, the door is indeed sufficiently opened. If this action cannot be performed (this is referred to as an action being *blocked*), we know the door is not sufficiently opened.

Given this notion of actions, we note that we may want to parametrize actions: if we are modeling communication of numbers, we do not want to introduce action *send0* to indicate we have sent a 0, *send1* for sending a 1, etc - we simply issue a $send(0)$, $send(1)$ and so forth, and define $send : \mathbb{N}$: an action that takes a natural number as parameter. Of course, we can also use multiple parameters: if we have $send : \mathbb{N} \times \mathbb{B}$, this indicates action *send* takes a natural number and a boolean value.

There is an important special action, the *deadlock* action δ, which illustrates that no action can be done and that there is no termination possible; once a process is in a deadlock state, it cannot do anything and will stay deadlocked forever. While it may seem a bit odd to introduce the notion of a deadlock in our model, it is generally not directly used within specifications but rather the result of other operators, the most important of which will be introduced in the next sections. The δ action has two distinct properties: if we can chose between a deadlock or another action, the other action will always be selected; this makes sense as we intend to avoid a deadlock as long as possible. Secondly, we cannot continue after a deadlock: the process is deadlocked, so there is nothing it can do. These properties are illustrated in axioms A6 and A7 (the precise meanings of $+$ and $\cdot$ will be illustrated in Section 3.6)

3.3 Multi-actions and communication

As we are modeling independent systems, it may be possible that multiple atomic actions are performed at exactly the same moment in time. This is denotes as $a \mid b \mid \ldots$: actions a, b, etc. all occur at exactly the same moment in time. While this may seem odd at first, remember that systems generally consist of multiple processes, and these processes are all independent so they can issue an action regardless of what another process is doing.

Using this concept of multi-actions, we can introduce the notion of *communication*: if one process performs a $send(1)$ action to indicate it sends the value 1, we want another process to be able to perform a $receive(1)$ action (but not $receive(2)$ for example, as value 2 clearly is not sent). This means we have to specify that actions *send* and *receive* in a sense 'belong' together: we want to combine them in a single action. This is achieved using the concept of a *communication operator*, which we call Γ_C: this Γ_C maps a multi-action to a new action, using the definitions as supplied in the set C. For example, we would define $C = \{send \mid receive \rightarrow communicate\}$ to illustrate that we refer to a *send* action simultaneously occurring with a *receive* action as a *communicate* action.

This means that if we apply communication set C to multi-action $send \mid receive$, we expect to obtain $communicate$, which is indeed the case since $\Gamma_{\{send \mid receive \rightarrow communicate\}}(send \mid receive) = communicate$, as illustrated by axiom C1. We note that if the actions have parameters, they are only allowed to communicate if all parameters are equal, and left as-is otherwise: for example, $\Gamma_{\{a \mid b \rightarrow c\}}(a(0) \mid b(0)) = c(0)$, but $\Gamma_{\{a \mid b \rightarrow c\}}(a(0) \mid b(1)) = a(0) \mid b(1)$, as illustrated by axiom C1.

The deadlock action δ deserves special mention when occurring in a multi-action: when δ occurs in some multi-action, this basically means that something is happening along with a deadlock. The result is, of course, that the entire process is deadlocked, as δ cannot be followed by an action. This property follows from axioms S1, S4 and C2.

3.4 Blocking actions

In most cases, we want to disallow certain actions to be issued. This may seem a bit counter-intuitive, as one may wonder why we introduced them to begin with. However, recall the previous section discussing communication: the communication operator only has an effect if all parameters are equal. Thus, if we have process P trying to send values by means of an action $send(n)$ and another process Q receiving values by means of $receive(m)$ where $send \mid receive \rightarrow communicate$, then we would end up with $communicate(n)$ if and only if $n = m$, and $send(n) \mid receive(m)$ otherwise. Clearly, we do not want to receive values that are not being sent - we want the result of $P \parallel Q$ to contain only $communicate(n)$ actions.

This can be prevented by using the *blocking operator* ∂_B, also known as the *encapsulation operator*. Generally speaking, the blocking operator replaces any action that is in the set B by a δ; this has the result of disallowing or blocking the action. For example, $\partial_{\{a\}}a = \delta$ and $\partial_{\{a\}}b = b$, as illustrated by axioms E2 and E3.

Using the blocking operator, if we specify $\partial_{\{send,receive\}}\Gamma_{\{send \mid receive \rightarrow communicate\}}P \parallel Q$, the result is that only $communicate(n)$ actions will 'survive' - the individual $send$ and $receive$ actions are blocked. As we first apply the communication operator Γ, the result is that any corresponding $send(n) \mid receive(n)$ actions will be replaced by $communicate(n)$ actions, after which any surviving $send$ and $receive$ actions are blocked.

3.5 Hiding actions

As might be expected, a system tends to grow rather large. In a sense, it will easily become cluttered with a lot of actions we are not interested in; for example, if we are creating a model of a coffee vending machine, we generally are not interested in all communication messages between the various processes. Rather, we are interested whether making a selection and inserting a number of coins always results in a cup of coffee being served or that it is possible for the system to deadlock at some moment in time due to an unforeseen sequence of events. This is where the internal action τ, also known as a *silent step* becomes interesting. We use τ to denote that the state of the process has changed, but there has not been any visible behavior.

In order to be able to apply this reasoning to our model, we introduce the *hiding operator* τ_I, which renames actions that occur in I to τ. The result is that, for example, $\tau_{\{a\}}a = \tau$ and $\tau_{\{a\}}b = b$, as illustrated by axioms H2 and H3. An important observation is to consider what the result is of a τ in a multi-action: actions are occurring, including internal behavior that is unobservable. One may expect that the result is that the τ action vanishes; if we derive $\tau_{\{a\}}a|b = \tau|b = b$, we see that this is indeed the case, as illustrated by axioms H4, H2 and S3.

3.6 Specification of processes

As the basics of processes have been covered in the previous sections, we will now focus on how processes are specified. We start by introducing the following operators:

- Alternative composition +
  The choice operator + specifies that there is a choice that needs to be made. For example, if we have $a + b$, either an a or a b action can be performed. Note that the choice is made in a non-deterministic manner: if both a and b can be done, a does *not* take precedence! Furthermore, if there is a choice between an action α and the deadlock δ, action α is always performed, since $\alpha + \delta = \alpha = \delta + \alpha$ as illustrated by axioms A6 and A1.

- Sequential composition $\cdot$
  The sequential operator $\cdot$ specifies that one action is followed by another action. For example, if we have $a \cdot b$, this illustrated that an action a must be followed by an action b.

- Condition operator $\rightarrow$
  The conditional operator $\rightarrow$ is similar to the if-then-else construct in standard programming languages. For example, $n > 4 \rightarrow a \diamond b$ specifies that if n is larger than 4, an a action is to be performed, otherwise a b action must be performed. We conveniently introduce $\alpha \rightarrow \beta$ as a shorthand notation for $\alpha \rightarrow \beta \diamond \delta$.

- Summation operator $\sum$
  The summation operator $\sum$ is basically a generalization of the + operator illustrated before, which we again shall introduce by using an example: $\sum_{i:\mathbb{B}} a(i)$ denotes that there is a choice of action $a(i)$ for any value $i \in \mathbb{B}$. Since $\mathbb{B} = \{true, false\}$, this means $\sum_{i:\mathbb{B}} a(i) = a(true) + a(false)$. However, this can be combined using the $\rightarrow$ operator as illustrated before: for example, if we specify $\sum_{i:\mathbb{N}} i < 10 \rightarrow a(i)$, this is equal to $a(0)+a(1)+a(2)+\cdots+a(8)+a(9)$ - the former is much more compact than the latter.

Processes are usually specified using recursion. For example process $P = a \cdot P$ is the process that can perform an a action, followed by an a action, etc. This makes it convenient to write down processes that do not terminate. Processes can have parameters as well: for example, if we declare $P(n : \mathbb{N}) = a(n) \cdot P(n + 1)$ then $P(0) = a(0) \cdot a(1) \cdot \ldots$: a process that performs an a action with value n and then increments this value n.

The behavior of a process is usually much clearer if we have a graphical way to illustrate processes, which we shall do by means of a Labelled Transition System or LTS. An LTS generally contains a set of states $\mathcal{S}$, an initial state $\mathcal{S}_0$, a set of actions $\mathcal{A}$ and a transition relation $\mathcal{R}$ that maps a state and an action to another state (thus, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$). An LTS is an ideal way to compactly visualize processes, as we shall show by illustrating some examples in Figure 3.1, where the initial state has an incoming arrow pointing into it.

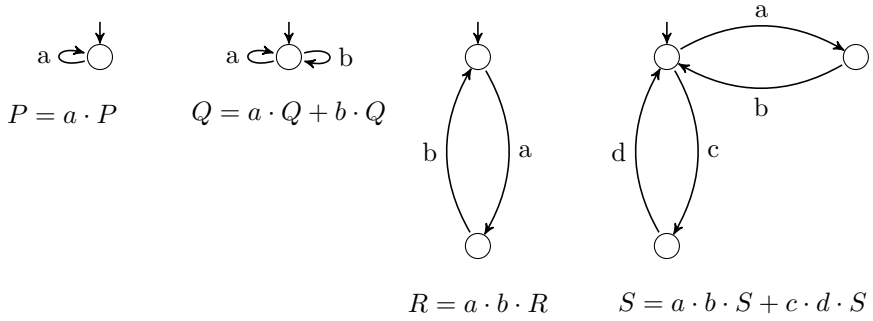


Figure 3.1: Examples of several LTS-es

3.7 The parallel operator

In the first section, we have somewhat informally described the $\|$ operator as an operator which given two processes results in the behavior of these two processes in parallel. In order to illustrate the way this works, let us consider the expected behavior of process $P \| Q$. There are three possibilities:

- Process P performs an action before process Q
- Process Q performs an action before process P
- Both processes perform an action simultaneously.

Let us illustrate this by considering a small example, consisting of process $P = a \cdot b$, process $Q = c$ and process $Z = P \| Q$. While we can formally derive process Z (this is performed in Appendix A), it is also possible to do so by simply following the three possibilities outlined above. We shall use these to construct process $Z = P \| Q$. First of all, note that the very first possibility is that process P performs an action, which results in an a action. Likewise, it's also possible that process Q performs an action, which is action c. Finally, both processes can perform an action simultaneously, which would result in the multi-action $a \mid c$. The result is illustrated in Figure 3.2 - we have numbered all states so we can refer to them.

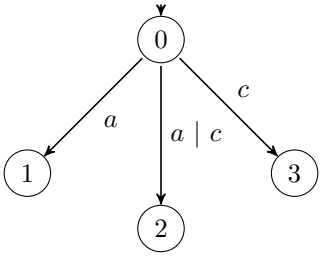


Figure 3.2: Under-construction LTS of process $Z = (a \cdot b) \| c$, step 1

Now, the next action can be performed. Let us consider the case in which process P has performed an a action. The only options are for process P to perform an action, which must be action b (since the a has already occurred). Process Q can also perform an action, which is action c. Finally, the b and c actions can be performed simultaneously, resulting in $b \mid c$. The result is illustrated in Figure 3.3.

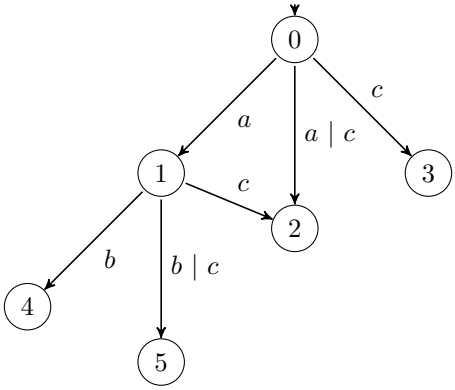


Figure 3.3: Under-construction LTS of process $Z = (a \cdot b) \| c$, step 2

There is an interesting observation to be made: in Figure 3.3, why does doing an c action in state 1 result in ending up in state 2 and not a new state? This question can also be rephrased: do we

end up in different states if we perform an a action followed by a c action compared to when we perform an $a \mid c$ action, e.g. we perform an a and c action simultaneously. This is not the case: after we have performed an a action in process P followed by a c action in process Q, process P can only do a b action while process c cannot do anything anymore; the same holds if we do a $a \mid c$ action. We continue by considering the next action after which process Q has performed a c action. Process Q cannot do anything anymore, so process P has to do an a action, as illustrated in Figure 3.4.

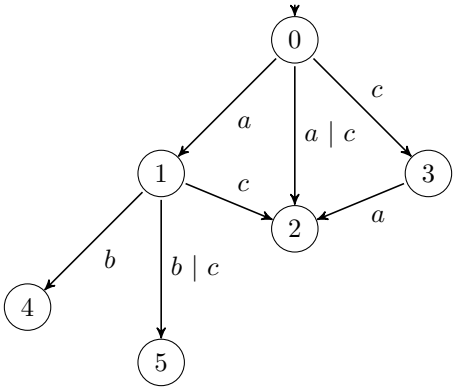


Figure 3.4: Under-construction LTS of process $Z = (a \cdot b) \parallel c$, step 3

We see that the reasoning above still holds in Figure 3.4: there is no difference between first doing an a action followed by a c action, first doing a c action followed by an a action or doing an a and c action simultaneously - regardless of the order we take them in, we end up in state 2. This diamond-like shape is actually very common when visualizing statespaces, and it is known as the notion of *confluence*: often, the order of two actions is not important as the resulting state is the same. This can lead to optimizations of the state space, as the whole state space doesn't need to be calculated in order to obtain this result. Finally, if we consider the remaining action that are still possible, we obtain:

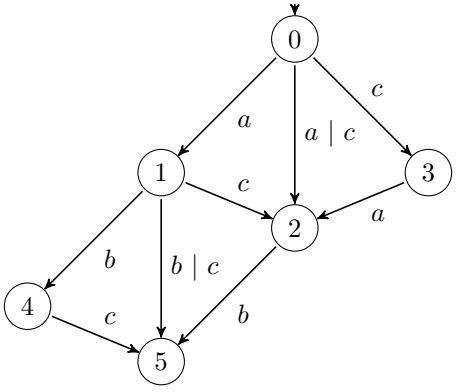


Figure 3.5: Completed LTS of process $Z = (a \cdot b) \parallel c$

As illustrated by Figure 3.2 to 3.5, even the parallel composition of two very simple processes results in quite a number of states. Often, these can be reduced by using the τ_I and ∂_B operators as illustrated in the previous sections. As derived in Lemma A.1, the resulting process of $(a \cdot b) \parallel c = a \cdot (b \cdot c + c \cdot b + b \mid c) + c \cdot a \cdot b + a \mid c \cdot b$, which is the same process as illustrated in Figure 3.5. The important notion here is that putting multiple processes in parallel results in a lot of previously unexpected possibilities, which is why formally modeling these processes is beneficial: conditions that are unexpected will always turn up sooner or later.

3.8 An example

In order to get a clearer view of this theory, let us consider the following example: suppose we have a very basic alarm clock, which starts sounding an alarm at 7:00 AM. Once it is sounding, the user has two options: the alarm can be switched off by pushing the off button or it can be postponed for 10 minutes by pushing the snooze button. This behavior is illustrated in the following algorithm:

Algorithm *AlarmClock*()
1. $alarmSounding \leftarrow false$
2. **while** $forever$
3. **do if** $alarmSounding$
4. **then** ▷ The alarm is sounding - handle buttons
5. **if** off button is pushed
6. **then** ▷ Alarm must be turned off
7. Turn the alarm off
8. $alarmSounding \leftarrow false$
9. **else if** snooze button is pushed
10. **then** ▷ We must delay the inevitable for 10 minutes
11. Turn the alarm off
12. Wait for 10 minutes
13. Turn the alarm on
14. **else** ▷ The alarm is not sounding
15. **if** it is 7:00 AM
16. **then** ▷ Wakey wakey!
17. Turn the alarm on
18. $alarmSounding \leftarrow true$

How have we constructed this algorithm? Generally, we have just followed our intuition and wrote down what we expect would work. Typically, the next task is to implement this system in some programming language and check whether it does what we intended. However, we will be using this algorithm to construct a process P of the system. First of all, we need to define the actions that we will be using. Keep in mind that actions can represent either an incoming or outgoing event, as illustrated in Section 3.2.

`AlarmOn`	Start sounding the alarm
`AlarmOff`	Turn the alarm sound off
`SnoozeButton`	The snooze button has been pushed
`OffButton`	The off button has been pushed
`Timer`	7:00AM has arrived
`Delay10`	10 minutes have passed

There is only a single variable in the algorithm, *alarmSounding*. This variable is used to see if we need to react on any buttons pushed, so our process will need this variable as well. This means our process will be $P(alarmSounding : \mathbb{B})$, and *alarmSounding* is initially $false$ due to the assignment in line 1. Let us focus on lines 5 - 8: if the alarm is sounding ($alarmSounding = true$), pushing the off button should stop the alarm sound. In our process, this would become:

$$alarmSounding \rightarrow \texttt{OffButton} \cdot \texttt{AlarmOff} \cdot P(false)$$

What does this represent? If *alarmSounding* holds (due to the conditional operator $\rightarrow$), attempt an `OffButton` action. If this is successful, it will be followed by an `AlarmOff` action and subsequently by process P where the parameter (this is the *alarmSounding* parameter) is $false$, as the alarm will no longer be sounding. This confirms with the algorithm lines 5 - 8. Subsequently, we can list the behavior of the snooze button similarly:

$$alarmSounding \rightarrow \texttt{SnoozeButton} \cdot \texttt{AlarmOff} \cdot \texttt{Delay10} \cdot \texttt{AlarmOn} \cdot P(alarmSounding)$$

Once again, we shall dissect this line piece by piece: if $alarmSounding$ holds, attempt a `SnoozeButton` action. If this is successful, we will in sequence perform `AlarmOff` - to switch the alarm sound off, a `Delay10` action to wait for 10 minutes, `AlarmOn` to start the alarm sound and finally, we return to our process without changing the parameter (since we just pass the same value of $alarmSounding$ to P again). This corresponds with lines 9 - 13. Finally, there is only the case left where the alarm needs to start sounding, which is:

$$\neg alarmSounding \rightarrow \texttt{Timer} \cdot \texttt{alarmOn} \cdot P(true)$$

If the alarm is not sounding, attempt a `Timer` action. If this is successful, continue by turning the alarm on by means of an `AlarmOn` action and continue our process while setting the $alarmSounding$ parameter to $true$. This confirms with the algorithm lines 15 - 18. We list the complete specification of process P:

$$\begin{aligned} &P(alarmSounding : \mathbb{B}) = \\ &\quad alarmSounding \rightarrow \texttt{OffButton} \cdot \texttt{AlarmOff} \cdot P(false) + \\ &\quad alarmSounding \rightarrow \texttt{SnoozeButton} \cdot \texttt{AlarmOff} \cdot \texttt{Delay10} \cdot \texttt{AlarmOn} \cdot P(alarmSounding) + \\ &\quad \neg alarmSounding \rightarrow \texttt{Timer} \cdot \texttt{alarmOn} \cdot P(true) \end{aligned}$$

Why do we claim this specification P is correct? First of all, note that the `OffButton`, `SnoozeButton` and `Timer` actions are always the first actions in a row to be performed in a conditional. This means, if the condition holds, we will attempt one of these actions. The intuition is that, suppose the user does not push the off button, the `OffButton` action is blocked. This means the first line will become:

$$alarmSounding \rightarrow \delta \cdot \texttt{AlarmOff} \cdot P(false)$$

In Section 3.3, we have seen that if we are able to chose between an action a or δ, we always chose action a since we want to avoid the deadlock. However, suppose all these actions are unavailable, then we would have no choice but to do a deadlock. Yet, there is an important aspect we have not illustrated so far: a process does not *have* to perform an action - this can be delayed. Usually, if only deadlock actions are available, we will attempt to wait until something will become available - which makes perfect sense since these actions are initiated by some event. More importantly, a deadlock will only show if all possible actions can never occur, which is usually what we wish to analyze.

Finally, we shall present an LTS of process P in Figure 3.6. We note that the behavior of the system is clearly visible in the LTS, especially compared to the algorithm.

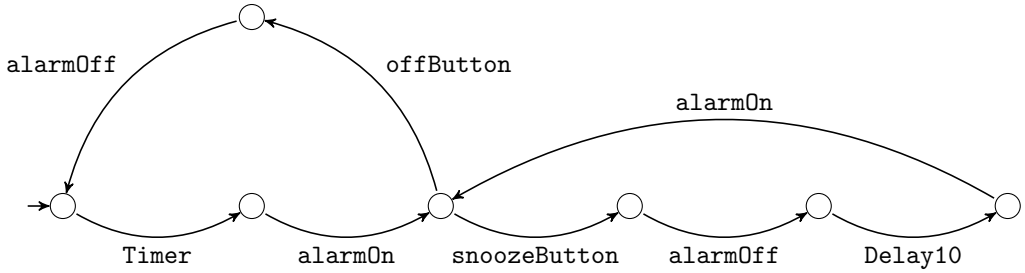


Figure 3.6: LTS of process P, the alarm clock example

Chapter 4

Operation

As outlined previously, all nodes are identical, their respective identification number being the only exception. How will a node know when it has to switch its light on or off and how are they kept synchronized? A standard way of dealing with such situations is to introduce the concept of a *leader*. This leader is responsible for initiating actions which should eventually lead to the desired behavior of the system. Initially, the leader must determine the properties of the network in use and relay necessary information to all other nodes in the system: each node has to know who the leader is, the node's coordinates in the network and what the actual size of this network is. Once the environment is properly set up, the leader will instruct all nodes which figure they should display. As each node knows the size of the network, its coordinates within this network and who the leader is, they will know whether they should switch their light on or off depending on the requested figure. The need for each of these properties will be elaborated in the next sections.

Since all nodes have pretty limited resources (these are discussed in Section 2.1), we intend to store as little information as possible. To this end, if a node knows the size of the network and its coordinates within this network, it can determine whether the light should be switched on or off - this removes the need for the leader to calculate per node if the light should be on or off; it can simply request a figure from all nodes and the nodes determine by themselves if the light should be turned on or off depending on their coordinates in the network, the network size and the figure to be displayed.

Based on this rough description, we can enumerate the steps needed in this system:

1. Leader election
   As all nodes are identical, they need to agree on who will be the leader. This leader is used to initiate any further communication.

2. Node coordinates
   The leader's first task is to determine the size of the network. However, before this can be done, every node must be assigned *coordinates*, which must be unique in the network. The leader considers itself to be at $\langle 0, 0 \rangle$, the other nodes are positioned relative to these coordinates and assume the same rotation as the leader node (i.e. the receiving node alters the meaning of its directions, resulting in it considering direction 0 to be the same direction as the leader node considers it to be)

3. Network size determination
   If all nodes have unique coordinates, we can determine the actual size of the network in use by determining the minimal and maximal x- and y-coordinates. This is necessary since a node initially has no idea what the size of the network in use is. We relay the minimum and maximum x- and y-coordinates throughout the network allowing every node to calculate

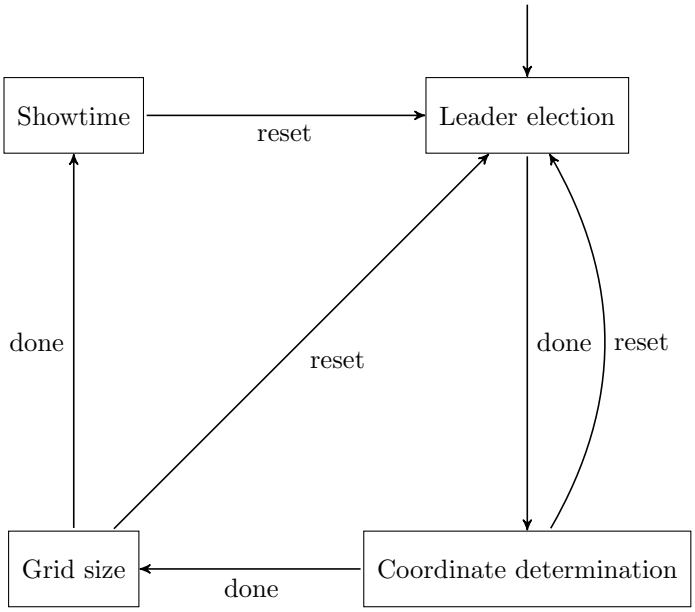


Figure 4.1: State transitions between the system steps

the size of the network. The combination of grid size along with the node's coordinates is enough to determine whether the light should be switched on or off for each figure.

4. Showtime
   Every node knows what it has to do to display the intended figure - the only catch is that nodes do not know when they have to change the figure. Since each node may run slightly faster or slower than the specified clock speed, we can't just let every node delay and change the figure - sooner or later, nodes will get out of sync this way. This is prevented by having the leader instruct all nodes when the time has come to alter the image that is currently being displayed.

Observe that this outline does not consider externally altered node configurations. In fact, we have no way of detecting whether a node is added or removed at all. However, we can detect whether there are multiple leaders, since this may result in contradictory messages being received - one leader may request that all nodes display the first figure, whereas the other leader request the second figure; this may result in nodes constantly changing figures. Thus, if such a condition arises, the first node that observes there is an contradiction within the network must issue a `reset` action to all neighboring nodes and resets itself, whereas nodes receiving this reset perform the same action (provided they have not already done so). The result is that a new leader is elected upon whom everyone agrees. To this end, we will also have to show that such a `reset` action cannot occur under ordinary circumstances.

We shall graphically illustrate the flow between these steps in Figure 4.1. Each of these steps will be discussed in subsequent sections.

4.1 Leader election

The very first operation is determining which node will become the leader. This is a well-known problem in distributed computing environments, for which many algorithms have been proposed. One of the first descriptions of such an algorithm can be found in [Lan77], where leadership election in an unidirectional ring is discussed. Let us first illustrate such a ring in Figure 4.2. In

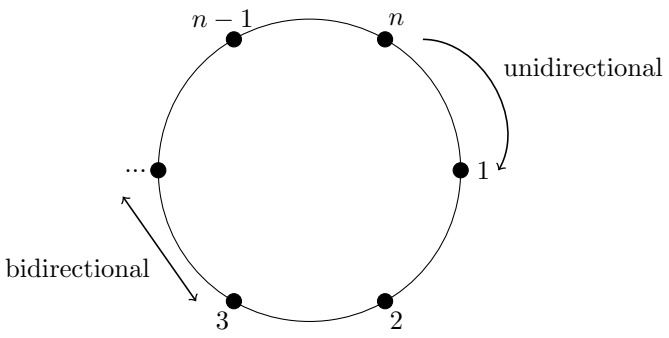


Figure 4.2: Structure of a ring network of n nodes

an unidirectional ring, each node i can only communicate with node $i+1$ and the final node n can only communicate with node 1; if the ring is bidirectional, node $i+1$ can also communicate with node i and node 1 can communicate with node n.

In the algorithm described in [Lan77], it is assumed that all nodes have a unique identifier with a total ordering. The result is a deterministic algorithm with a running time of $\mathcal{O}(n^2)$ which can be improved to $\mathcal{O}(n \log n)$ [CR79]. We note that construction of a deterministic algorithm is only possible if nodes have such a unique identifier [Lyn96, Theorem 3.1]. If such a unique identifier is not present, it is possible to solve the problem using a randomized algorithm as discussed in [IR90].

We will first discuss the relationship between our node networks, which we shall refer to a *grid*, and the ring networks described in the literature. Note that unlike an ordinary grid, nodes in our grid network are only allowed to communicate with adjacent nodes.

Lemma 4.1. *Let G be a $n \times n$ grid. Then, there is no general way to transform G to a ring containing $n * n$ edges.*

Proof. If we consider some 3×3 node configuration as G, we observe there is no way to transform G to a 9-edge ring: this is due to the fact that G contains a node in the center, which prevents construction of such a ring as the grid only allows horizontal and vertical connections between nodes. This is illustrated in Figure 4.3, where the thick lines illustrate the two neighbors of each node. As can be seen, the node in the center cannot be connected without removal of some other node.

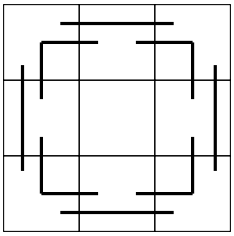


Figure 4.3: Attempt to transform 3×3 grid G to a ring

Thus, since constructing a ring is already impossible for this specific grid G, it cannot be done in general as claimed. □

The previous lemma leads to an interesting observation: since a 3×3 grid G cannot be transformed into a ring of 9 edges, there cannot be a ring describing this specific 3×3 grid G. We obtain:

Corollary 4.2. *Let R be a ring containing $n*n$ edges. Then, there is no general way to transform R to a grid of size $n \times n$.*

From the two observations above, we conclude that we cannot use algorithms operating on ring networks. However, the study of ring networks is still useful to us: we can show that if leader election without unique identification numbers is not possible for ring networks, it cannot be done for grid networks either.

Theorem 4.3. *There is no deterministic algorithm $\mathcal{A}$ for leader election for any grid G containing $n \times n$ edges if all nodes are completely identical.*

Proof. Let us consider a grid G of 2×2 nodes. Clearly, we can transform this grid to a ring as it is basically already a ring of 4 edges. From [Lyn96, Theorem 3.1] it follows that this problem cannot be solved without a way to distinguish the individual nodes, proving the theorem. □

Based on the hardware in use as outlined in Section 2.1, we expect to have a pretty poor pseudo random generator. Based on this observation and Theorem 4.3, it follows that the node identifiers are actually required. Fortunately, there is a unique identifier per node and we consequently use this identifier to elect a unique leader: the node with the lowest identifier becomes the leader. The following algorithm is proposed to carry out this procedure:

Algorithm *LeaderElection*(id)
1. ▷ Initially, we believe we are the leader - but we haven't told anyone yet
2. $mustSend \leftarrow true$
3. $leaderid \leftarrow id$
4. **while** leader election is in progress
5. **do if** $mustSend$
6. **then** ▷ We haven't yet told who we believe the leader is
7. Send a message $(id, leaderid)$ to all directly connected nodes
8. $mustSend \leftarrow false$
9. **if** a message $(nodeid, newid)$ is received
10. **then** ▷ Someone believes he has new leader for us
11. **if** $newid < leaderid$
12. **then** ▷ The proposed leader is better than the one we currently have
13. $leaderid \leftarrow newid$
14. ▷ Inform all our direct nodes about this new leader
15. $mustSend \leftarrow true$
16. **else** ▷ Our leader is better than what we received
17. **ignore**
18. ▷ We are done. If $id = leaderid$, we act as leader

First of all, line 4 states that this algorithm should run while the leader election is in progress. This obviously begs the question: when is the leadership election done? The answer is that we cannot know - a leader does not keep an overview of all nodes, so it can't keep track of whether all nodes acknowledge their leader. And even if such a mechanism was to be implemented, if some node fails and never acknowledges the leader, we'd be waiting indefinitely. To prevent these issues, we say that the leader election phase ends if updates have not been received for 2 seconds. Even if the clock speed of all nodes may be off by a small percentage, the assumption is that messages are processed fast enough so that any lower identifier would have been received within this time.

By visual inspection of this rather straightforward algorithm, we expect it does the right thing: out of all nodes, the node with the lowest id becomes the leader. Since all id's are distinct, we know there is only one lowest id. Thus, a single node becomes the leader and all other nodes accept this course of action. However, can we provide a better founded argument? The algorithm will be

described by means of a model in Chapter 6, where it is proved that the necessary properties are satisfied.

4.2 Coordinates determination

Once a leader has been chosen, it must assign every node unique coordinates. The leader will consider itself to be at $\langle 0, 0 \rangle$ and in turn assign coordinates relative to this position to its neighboring nodes. However, nodes can be rotated any multiple of 90°, so it is important that all nodes in the network consider 0 as the same direction, regardless of how each individual node is rotated. To this end, we must include the direction where the request should be received in the message, allowing the receiving node to determine how it is rotated and subsequently alter the meaning of its directions as needed. This results in all nodes assuming the rotation of the leader. Let us introduce how we can calculate the rotation based on the incoming and desired direction. Since all nodes are square-sized, they can be rotated $0°, 90°, 180°$ and $270°$. We shall introduce r as a variable indicating the angle at which the node was rotated, where $rotation = r * 90°$ to the right. The result is illustrated in Figure 4.4.

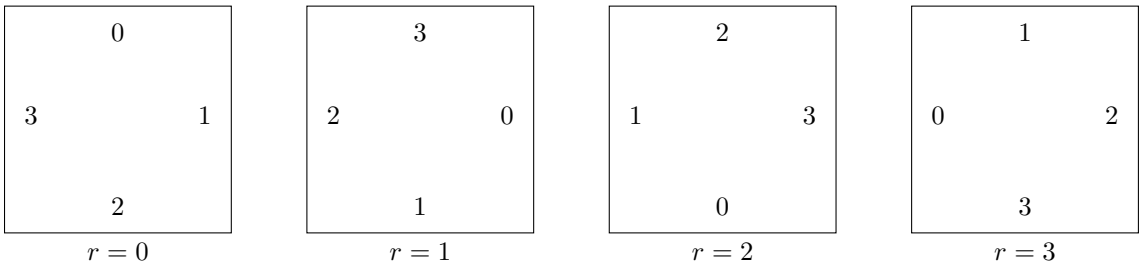


Figure 4.4: Node rotation configurations, where $rotation = r * 90°$

By looking at Figure 4.4, we see that if an unrotated node ($r = 0$) is sending to direction 1, a node with $r = 1$ would be sending to direction 0, a node with $r = 2$ would be sending to direction 3 and a node with $r = 3$ would be sending to direction 2. There is a clear pattern here, which makes much more sense when we describe it: if some node $\mathcal{N}$ which is rotated r turns to the right sends to direction d, this is the same as if an unrotated node would send to direction d rotated r turns to the left: the end result is the same.

By the above reasoning, we can define a rotate-left function, $rl(d, r) = (d - r) \bmod 4$: if the rotation is r and the direction is d, the unrotated direction will be $d - r$, and since we have 4 directions, we must perform the calculation modulo 4. This function can also be used to find the actual rotation of a node: if we know the incoming direction d and the expected direction e, we are interested in how much d should be rotated to match e - that is, we want to know what $e - d$ is, which is what $rl(e, d)$ calculates.

Once a node is aware of its rotation r, it can use this rotation to align its directions with the unrotated case. Let us consider an example: if a node has rotation $r = 2$ and it intends to send to unrotated direction $d = 1$, it should send to direction 3, as illustrated in Figure 4.4. As we have defined the rotation r as the number of turns to the right, we can simply use output $(d + r) \bmod 4$ - this makes sense, as we are countering the left turns introduced by the rl function. To this end, we appropriately introduce a rotate-right function $rr(d, r) = (d + r) \bmod 4$.

We need but one function: in order to determine the rotation of a node, we need the direction in which a message was received as well as the direction this message should have been received in. The first is available by means of the hardware (refer to Section 2.2), but the intended direction must be embedded in the message. Let us consider $r = 0$ in Figure 4.4: if we send a message to

direction 0, we want it to be received in direction 2. To this end, we can define a receiving-direction function $rd(i) = (i + 2) \bmod 4$.

Finally, we must define the coordinate system in use. To this end, we define the bottom-left coordinate as the lowest coordinate and subsequently, top-right is the highest coordinate in use. This is illustrated in Figure 4.5.

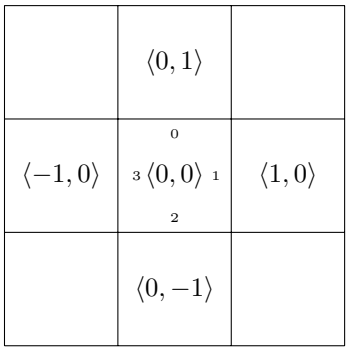


Figure 4.5: Relative coordinates between nodes

Based on Figure 4.5, we can define a function $\Delta(c, d)$, which given a coordinate $c = \langle x, y \rangle$ and a direction d returns the corresponding coordinate if coordinate c is moved one position in direction d:

d	$\Delta(\langle x, y \rangle, d)$
0	$\langle x, y + 1 \rangle$
1	$\langle x + 1, y \rangle$
2	$\langle x, y - 1 \rangle$
3	$\langle x - 1, y \rangle$

Using these definitions, we propose the following algorithm:

Algorithm *DetermineCoords*$(id, actAsLeader)$
1. ▷ Initially, we always believe we are at $\langle 0, 0 \rangle$ and we are not rotated
2. $coord \leftarrow \langle 0, 0 \rangle$, $r \leftarrow 0$
3. ▷ The leader must initiate the procedure
4. $mustSend \leftarrow actAsLeader$
5. **while** coordinate determination is in progress
6. **do** ▷ Communicate new coordinates as necessary
7. **if** $mustSend$
8. **then** ▷ Communicate coordinates to direction i
9. For all $0 \leq i \leq 3$: send coordinates $\Delta(coord, rr(i, r))$ using output i
10. where receiving direction should be $rr(rd(i), r)$
11. $mustSend \leftarrow false$
12. **if** a message $(newcoord, dir)$ has been received from direction $source$
13. **then if** $coord = \langle 0, 0 \rangle \wedge \neg actAsLeader$
14. **then** ▷ We did not receive initial coordinates, so update them
15. $coord \leftarrow newcoord$
16. $r \leftarrow rl(dir, source)$
17. ▷ We must inform our neighbors
18. $mustSend \leftarrow true$
19. **else** ▷ We already know our coordinates
20. **if** $coord \neq newcoord$
21. **then** ▷ We receive different coordinates - this indicates multiple leaders
22. **reset**

```
23.                 else ▷ Position is correct - check rotation
24.                     if rl(dir, source) ≠ r
25.                       then ▷ Rotation is not correct
26.                             reset
27.                       else ▷ We received the same coordinate on the correct input
28.                             ignore
```

First of all, it must be noted that there are two approaches possible while sending updates in line 7 - 11: we can either send to actual output pin i, regardless of its orientation or we can send to the pin that becomes output pin i after countering the rotation. In our approach, we have chosen the first option as it makes the model clearer to understand and the second option doesn't provide any additional benefit.

The loop in line 5 has to terminate at some point - however, as before, we do not know when all nodes have been given coordinates as we do not know how many nodes there are. Thus, our only option is to continue to the next phase if we haven't received messages for a period of time, similar to leadership election.

Using the proposed algorithm, we note that once the rotation r is calculated as $r = rl(d, n)$, we expect that sending a message back to n using rotation r by means of $rl(n, r)$ means we actually use direction d. This is proved in Lemma 4.4.

Lemma 4.4. *For any node number n and direction d, it holds that $rr(n, rl(d, n)) = d$*

Proof.

$$
\begin{aligned}
& rr(n, rl(d, n)) = d \\
= & \{ \text{ Declaration } rr \ \} \\
& (n + rl(d, n)) \bmod 4 = d \\
= & \{ \text{ Declaration } rl \ \} \\
& (n + ((d - n) \bmod 4)) \bmod 4 = d \\
= & \{ \text{ Algebra } \} \\
& (n + d - n) \bmod 4 = d \\
= & \{ \text{ Algebra } \} \\
& d \bmod 4 = d \\
= & \{ \ 0 \leq d \leq 3 \ \} \\
& true
\end{aligned}
$$

□

An interesting observation is that Lemma 4.4 actually implies the condition in algorithm line 24 cannot occur: if the leader sends a message from which the rotation is calculated, subsequent verification of the calculated rotation will always succeed. However, it is not specified whether nodes can be rotated without resetting them; if this behavior is possible, algorithm line 24 can indeed occur and the result is that the system will reset itself as intended.

4.3 Grid size determination

Once a node has entered this phase of the protocol, the node knows whether it is the leader, its coordinates within the grid and its rotation relative to the leader. However, the size of the grid is not known, while it is needed in order to determine which light should be switched on/off to display the intended figure. To this end, the grid size determination algorithm is run: each node communicates its maximal and minimal coordinates to its neighbors, which compare these

coordinates to their current minimal and maximal coordinates. If this results in better (i.e. smaller minimum or greater maximum) coordinates, the node will update their current coordinates and communicate these to all adjacent nodes. The desired result is that every node learns the minimal and maximal coordinates in this grid, from which the total grid size can be derived.

First of all, we have to determine how coordinates are compared: when is one coordinate smaller or greater than another coordinate? In order to do this, keep in mind that we are looking for the bottom-left and top-right coordinates. In other words, coordinates $\langle x_1, y_1\rangle \leq \langle x_2, y_2\rangle$ if $x_1 \leq x_2$ and $y_1 \leq y_2$. The same goes for the maximum coordinates: $\langle x_1, y_1\rangle \geq \langle x_2, y_2\rangle$ if $x_1 \geq x_2$ and $y_1 \geq y_2$. This means the minimum of two coordinates are their smallest x and y values, whereas the maximum of two x and y values are their largest x and y values. Thus, $\min(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle) = \langle \min(x_1, x_2), \min(y_1, y_2)\rangle$ and $\max(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle) = \langle \max(x_1, x_2), \max(y_1, y_2)\rangle$.

Using these declarations, we propose the following algorithm to perform these actions:

Algorithm *DetermineGridSize*($id, actAsLeader, leaderid, coord$)
1. ▷ Initially, we believe our own coordinates are both the minimum and maximum
2. $minCoord, maxCoord \leftarrow coord, coord$
3. ▷ If we are leader, initiate the procedure. Otherwise, just wait
4. $mustSend \leftarrow actAsLeader$
5. **while** grid size determination is in progress
6. **do** ▷ Communicate new min/max coordinates as necessary
7. **if** $mustSend$
8. **then** ▷ Broadcast the min/max coordinates
9. Send $(leaderid, minCoord, maxCoord)$ to all outputs
10. $mustSend \leftarrow false$
11. **if** a message $(lid, minc, maxc)$ has been received
12. **then** ▷ Received a new message - check consistency
13. **if** $lid > leaderid$
14. **then** ▷ There are multiple leaders!
15. **reset**
16. **if** $minc < minCoord$ **or** $maxc > maxCoord$
17. **then** ▷ We have obtained smaller minimal or greater maximal coordinates
18. $minCoord \leftarrow \min(minc, minCoord)$
19. $maxCoord \leftarrow \max(maxc, maxCoord)$
20. ▷ Update our leader id; lid can only be less or equal to
21. ▷ the current leader id
22. $leaderid \leftarrow lid$
23. ▷ Inform our neighbors about this
24. $mustSend \leftarrow true$
25. **else** ▷ We have a clearer picture of our grid than our neighbor. Yet,
26. ▷ we may have received an even lower leader id so honor it
27. $leaderid \leftarrow lid$

As usual, we should define what the condition in line 5 is: just looping forever waiting for new coordinates that never arrive can hardly be considered productive. However, like during the leader election, we cannot know if we have received the 'correct' minimum/maximum coordinates, since each node cannot know what the network's minimum/maximum coordinate is - this is why we introduced this phase to begin with! Thus, all we can do is wait until we have not received updates for a certain period of time and assume the maximum coordinates has been received by then, similar to what we do in the leadership election phase.

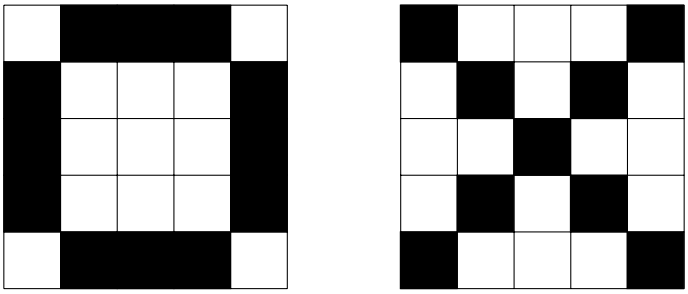
4.4 Showtime

During this phase, all necessary information has been acquired: each node knows who the leader is, its coordinates within the grid and rotation relative to the leader and finally, the size of the grid in use. The only remaining part is that nodes should display the desired figures; these actions are initiated by the leader node. We assume that upon power-on, the light is switched off, since there is too little information available to start displaying a figure.

The following algorithm is proposed:

Algorithm *Showtime*$(id, actAsLeader, leaderid, lpos, mincoord, maxcoord)$
1. **while** true
2. **do** ▷ The leader is responsible for initiating activity
3. **if** $actAsLeader$ **and** 2 seconds have elapsed
4. **then** ▷ The time has come
5. $figure \leftarrow (figure + 1) \bmod 2$
6. Update light to confirm to $figure$
7. Send a message $(id, figure)$ to all directly connected nodes
8. **if** a message $(lid, newfigure)$ has been received
9. **then if** $lid > leaderid$
10. **then** ▷ Someone else is pretending to be our leader
11. **reset**
12. **else if** $figure \neq newfigure$
13. **then** ▷ Follow the leader
14. $figure \leftarrow newfigure$
15. Update light to confirm to $newfigure$
16. ▷ We may have received an even lower leader id, so honor it
17. $leaderid \leftarrow lid$
18. Send a message $(leaderid, newfigure)$ to all directly connected nodes
19. **else** ▷ Our current status is fine, but the leader id may be lower
20. $leaderid \leftarrow lid$

We note that we need a function which, given a node's coordinates, the minimal/maximal node coordinate and the figure that needs to be displayed determines whether the light should be switched on or off. We shall first give a specification for figures 0 and 1 as illustrated in Figure 4.6.



$n = 0$ $n = 1$

Figure 4.6: Figures displayable by the nodes

The circular figure, $n = 0$, can be shown by lighting only the nodes at the border, with the exception of the nodes at the far edges. This gives the following formula:

$$light(\langle node_x, node_y\rangle, \langle min_x, min_y\rangle, \langle max_x, max_y\rangle, \text{circle}) = \\ (node_x = min_x \vee node_x = max_x \vee node_y = min_y \vee node_y = max_y) \wedge \\ \langle node_x, node_y\rangle \notin \{\langle min_x, min_y\rangle, \langle max_x, min_y\rangle, \langle min_x, max_y\rangle, \langle max_x, max_y\rangle\ \}$$

As for the cross, determining the light state is a bit more involved. First of all, we want to normalize the grid coordinates, so that coordinate $\langle 0, 0\rangle$ is the bottom-left. This is illustrated in Figure 4.7. Note that $min = \langle -2, -1\rangle$ and $max = \langle 0, 1\rangle$ - to this end, we need to subtract min from all coordinates.

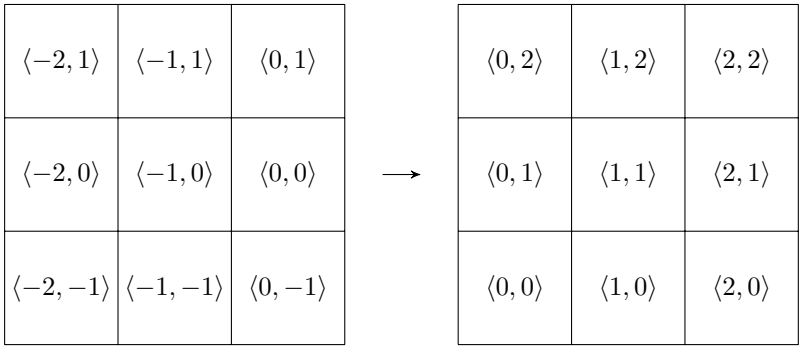


Figure 4.7: Coordinate normalization: bottom-left becomes $\langle 0, 0\rangle$

Now, we note that the bottom-left to top-right diagonal is simply determining whether the x and y coordinates are equal. The top-left to bottom-right coordinate means that the sum of the x and y coordinate is a fixed value: namely, the dimension of the grid minus 1. This gives the following formula:

$$light(\langle node_x, node_y\rangle, \langle min_x, min_y\rangle, \langle max_x, max_y\rangle, \text{cross}) = \\ x = y \vee \\ x + y = d \\ \textbf{where} \quad \langle x, y\rangle = \langle node_x - min_x, node_y - min_y\rangle \\ d = max_x - min_x$$

4.5 Resetting the system

The previous algorithms all rely on a `reset` action, which serves the purpose of instructing the complete node network to reset itself as something is flawed. Obviously, we need to describe this operation as we definitely do not intend to reset the system forever. To this end, upon power-on, a node must initialize $mustReset = false$ and $resetcount = 0$.

Algorithm *ResetAlgorithm*()
1. **while** *true*
2. **if** *mustReset*
3. **then** ▷ We haven't sent our reset to our neighbors yet
4. Send a message *resetcount* to all directly connected nodes
5. ▷ Increment the reset count, to ensure we do not honor this reset again
6. $resetcount \leftarrow (resetcount + 1) \bmod 10$
7. Reset our current node
8. **if** a message *count* is received
9. **then** ▷ We might have to reset

```
10.        if count ≥ resetcount
11.          then ▷ We must be resetting
12.               mustReset ← true
13.          else ▷ We have already honored this reset round
14.               ignore
```

As can be seen, the *ResetAlgorithm* algorithm runs forever - this is because the algorithm should always be active, as reset messages must be processed in any state of the system: *ResetAlgorithm* must always run in parallel with the current algorithm in the system. Furthermore, the result of the `reset` actions is that *mustReset* should be set to *true*.

The actual reset performed in line 7 should perform a reinitialization of the node. Specifically, the node should start in leader election phase, while *mustReset* is set to $false$. The *resetcount* value must not be changed, as this value is used to keep the network from resetting indefinitely.

4.6 Research questions

We will describe several research questions which will be answered in this thesis:

1. The system sketch describes distinct system modes and transitions between these modes. What is the system behavior if these modes are merged, i.e. any message can arrive in any mode?
   This question is discussed in Sections 10.1 to 10.3.
2. Does the network adapt correctly if nodes are added or removed from the system?
   This question is discussed in Section 10.5.
3. Can we determine how long a node should wait before electing itself as leader?
   This question is discussed in Section 11.3.
4. What happens if messages are not guaranteed to arrive?
   This question is discussed in Section 10.4.

We will begin by proving correctness of the algorithms in a fixed configuration, under ideal circumstances: messages always arrive and the node configuration is never changed. The goal is to determine whether the algorithms we have presented are correct at all. In Chapter 10, the influence of unreliable communication and adding/removing nodes will be discussed.

Chapters 6 to 9 all discuss the following 3×3 grid configuration:

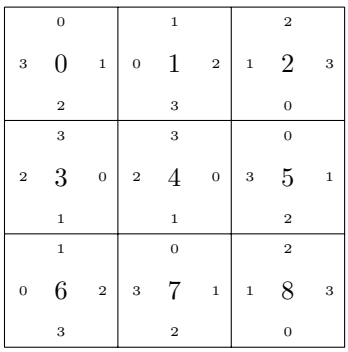


Figure 4.8: The 3×3 node configuration used during the analysis in chapters 6 to 9

Unfortunately, it is not possible to model an arbitrary configuration, as the statespace tends to grow so large in such situations that it is no longer manageable. Fortunately, a 3×3 grid seems

sufficient to determine problems within the system; after completing the analysis of a 3×3 grid, an attempt has been made to analyze a 4×4 grid. The results were that even though the analysis took much longer, it did not yield any different results. This leads us to believe that analyzing a 3×3 grid is sufficient as it already illustrates any potential problems.

Chapter 5

General notes on the model

As outlined in the overview in Chapter 2, each node can only interact with four adjacent neighboring nodes. However, this has to be made explicit in our model: we need to define actions to interact between such nodes. This means we need some way to ensure whenever nodes interact, they can only do so with adjacent nodes and not just any node in the network. First of all, we define the size of the network as $\mathcal{D} \times \mathcal{D}$. In order to make this easier, we sequentially number the nodes, from $0 \ldots N-1$, where $N = \mathcal{D}^2$, the total number of nodes in the network. The idea is, that if we simply number nodes sequentially, we can determine whether nodes can communicate with each other based simply on their number. Note that we *cannot* use node coordinates to determine whether this node can reach another node, since these coordinates are determined by an algorithm later on. Thus, the only purpose of the node number is to model the relationship between a node and its neighbors; it is different from the node's internal identification number. More on this distinction will be discussed later.

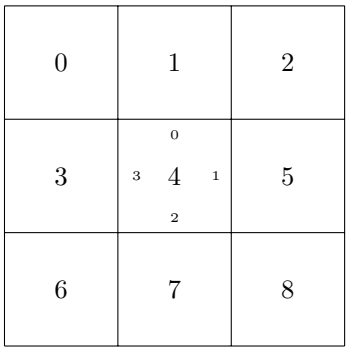


Figure 5.1: General setup of unrotated nodes in a 3×3 grid, thus $\mathcal{D} = 3$

Let us consider node number 4 in the setup in Figure 5.1: this node can reach node number 1 using direction 0, node number 5 using direction 1, node number 7 using direction 2 and finally node number 3 using direction 3. We observe that the following relationship between node number n and direction d holds:

Direction	**Node number**
0	$n - \mathcal{D}$
1	$n + 1$
2	$n + \mathcal{D}$
3	$n - 1$

However, just this relation is not enough: we have to explicitly disallow, for example node number

2 communicating in direction 1. By the above table, node number 2 is able to communicate with node number 3, which is not possible in reality. Therefore, we introduce extra constraints: nodes at the left edge of the grid cannot communicate in direction 3, and likewise for nodes at the right edge in direction 1, the top edge in direction 0 and the bottom edge with 2. Based on this specification, we define $neighbornum(n, d)$, which returns the node number of node number n's neighbor in direction d as:

$$neighbornum(n,d) = \begin{cases} n-\mathcal{D} & \text{if} \quad d = 0 \wedge y > 0 \\ n+1 & \text{if} \quad d = 1 \wedge x < \mathcal{D}-1 \\ n+\mathcal{D} & \text{if} \quad d = 2 \wedge y < \mathcal{D}-1 \\ n-1 & \text{if} \quad d = 3 \wedge x > 0 \end{cases}$$

$$\text{where} \quad \begin{array}{l} x = n \bmod \mathcal{D} \\ y = n \operatorname{div} \mathcal{D} \end{array}$$

Note that there are cases when we deliberately left *neighbornum* unspecified: if such a condition arises, the node in question simply does not have a neighbor in the given direction.

Using this *neighbornum* function, we can discuss how we send messages between nodes. Obviously, we cannot just tell messages to arrive - so we have to introduce actions for this purpose. To this end, each phase uses a `send(num,...)` action, which sends the specified parameters to a node number *num* whereas an action `recv(num,...)` receives such a message. These actions are combined using communication action `comm(num,...)` - thus, if a process wishes to perform a `send` action, there must be another process capable of doing a `recv` action with identical parameters. However, this has an important consequence: since it is a communication action, every `send(num,...)` must have a corresponding `recv(num,...)` action in another process - which means we cannot simulate sending messages to nodes that do not exist, since such actions will be blocked. In reality, the hardware will just put signals on pins that are not connected to anything and since this does not change the internal state, this makes no difference.

If a message needs to be broadcast to all four adjacent nodes, it is safe to assume that this message is sent in a specific order: that is, the message will first be sent to direction 0, then to direction 1, etc. Thus, we introduce a process variable s which contains the number of the direction we need to send to: if $s = 0$, we have to send to direction 0, so once that has been accomplished, we increment s as we now need to send to direction 1. We repeat this for all directions, so if $s = 4$, this indicates we have sent to all directions. Should we subsequently need to send messages again, we can simply set $s = 0$.

Why do we need the distinction above? One may expect that just using four consecutive `send` actions will have the same effect, and this is true if a node has four neighbors. If it does not, at least one of the `send` actions will be blocked, as there is no corresponding `recv` action. Clearly, this is not what we want, so we introduce an `intern` action, which simulates putting a signal on a pin which will not be received. The result is that in our model, if a message needs to be sent to direction s, there are two possibilities: if there is indeed a neighbor at that direction, we use the `send` action. If there is no neighbor, we use the `intern` action to simulate that the node is sending a message but nobody receives it. Conveniently, we can re-use this notation in order to model the loss of messages, as we shall discuss later.

As stated in Section 4.1, the node with the lowest unique identifier should become leader. To this end, we introduce a value $\mathcal{L}$, which is the number of the node that obtains identification number 0 - and since all identification numbers are natural numbers, this is the lowest identification number and thus the leader node. We need to define a function to map a node's number to an unique identifier in such a way that the node which number equals $\mathcal{L}$ will obtain id 0, whereas all other nodes will obtain unique id's that are greater than 0; this is desirable as we want to be able to check the model for boundary conditions: we are not only interested in the case where the node with the lowest identification number is at the top-left of the network, but also what

happens when we place it somewhere in the center of our network. To this end, we define a function $mapNodeId(num)$, which maps node number num to a node id in such a way that $mapNodeId(\mathcal{L}) = 0$ and all other numbers are distinct and greater than 0. We use declaration $mapNodeId(num) = (num - \mathcal{L}) \bmod \mathcal{D}^2$, as it satisfies the specification above.

Since the required `send`, `recv` and `comm` actions are different in each phase, we will not discuss their declarations here. However, the grid-positioning is always needed throughout the algorithms, so the corresponding mCRL2 declarations will be discussed now:

```
1  sort    Direction = Nat;
2          Coord = struct coord(x:Int,y:Int);
```

The first line declares the *Direction* type, which is just a natural number, as it is in the previous discussions. The reason we declare it as a new type is to make it clearer in the models when we are dealing with a direction. The next line is the declaration of the *Coord* type - a single *Coord* is a combination of two integer values, the first being the x coordinate and the latter being the y coordinate.

```
3  map     DIM: Pos;
4  eqn     DIM = 3;
5
6  sort    RList = List(Nat);
7  map     RotationList: RList;
8  eqn     RotationList = [ 0, 3, 2, 1, 1, 0, 3, 0, 2 ];
```

Since we always need the grid dimensions, we introduce a function DIM which simply returns $\mathcal{D}$, the dimensions of the grid - in this case, a 3×3 grid. The same holds for the rotations: we use a list *RotationList* containing $\mathcal{D}^2$ natural numbers. Each number denotes the number of right-rotations of the specified node. Thus, using the definition of *RotationList* as above, node number 1 would be rotated $3 * 90° = 270°$. The reasoning of using such a list is that once a node n sends a message to direction d, we can use this list to transform direction d based on the rotation of node n. If we introduced a parameter for each node indicating how much it is rotated, a node has information about its rotation. However, our goal is to introduction rotation as a malign force of nature: a node does not know whether it is rotated or how much, it just is, and it is up to the algorithms to decide how much this rotation actually is, as it is not possible for a node to circumvent the rotation.

In order to illustrate how this works, we first introduce the rotate-left and rotate-right functions rl and rr as discussed in Chapter 4.2:

```
9  map     rr: Direction#Int -> Nat;
10 var     d:Direction;
11         n:Int;
12 eqn     rr(d,n) = (d+n) mod 4;
13
14 map     rl: Direction#Direction -> Nat;
15 var     d,e: Direction;
16 eqn     rl(d,e) = (d-e) mod 4;
```

Then, using the *RotationList* and the rr function, we can introduce the concept of rotated nodes in our model: whenever node number num intends to send to direction d, we should use output direction $rr(d, RotationList.num)$. So subsequently, the number of the node receiving the message is $neighbornum(num, rr(d, RotationList.num))$. This means the rotation of a node will always apply, and more importantly, this rotation is fixed.

```
17 map     neighbornum: Int#Direction->Int;
18 var     n:Int;
19 eqn     neighbornum(n,0) = if(n div DIM>0,n-DIM,n);
20         neighbornum(n,1) = if(n mod DIM<DIM-1,n+1,n);
21         neighbornum(n,2) = if(n div DIM<DIM-1,n+DIM,n);
22         neighbornum(n,3) = if(n mod DIM>0,n-1,n);
```

We define the *neighbornum* function as illustrated previously, with one major difference. Previously, if there was no neighbor for node n in direction d, we would leave $neighbornum(n, d)$ unspecified. However, in our models, we need to determine whether there is such a neighbor as we need to distinguish whether to use a `send` or `intern` action. Thus, we introduce an if statement: if node n has a neighbor in direction d, $neighbornum(n, d)$ will return this neighbor. If there is no such neighbor, $neighbornum(n, d) = n$.

```
23 map     move: Direction#Coord->Coord;
```

We declare the *move* function, which takes direction d and coordinate c. The goal is to return coordinate c moved one position in direction d, which is equivalent to $\Delta(c, d)$ as illustrated previously.

```
24 var     x,y: Int;
25 eqn     move(0,coord(x,y)) = coord(x,y+1);
26         move(1,coord(x,y)) = coord(x+1,y);
27         move(2,coord(x,y)) = coord(x,y-1);
28         move(3,coord(x,y)) = coord(x-1,y);
```

We introduce two dummy integer variables x and y. For each direction, we specify the return value of the function, similar to how this is done in Figure 4.5 and the corresponding table.

```
29 map     LEADERNUM: Pos;
30 eqn     LEADERNUM = 5;
31
32 map     mapNodeId : Nat -> Nat;
33 var     n:Nat;
34 eqn     mapNodeId(n) = (n - LEADERNUM) mod DIM*DIM;
```

We have to specify which node will become the leader node. To this end, $LEADERNUM$ is equal to $\mathcal{L}$, the number of the node that should become leader. Finally, we define the $mapNodeId$ according to the definition given previously.

Chapter 6

Leadership election

Before we discuss any model, we have to consider a very important question: what exactly do we want to show? The goal of any model is to determine whether the system does 'what it should do', so let us begin by defining what it should do: our goal is, to determine whether the leadership election algorithm as outlined in Section 4.1 always causes exactly one node to consider itself as a leader and whether all other nodes agree on the same leader. As outlined previously, we will only consider ideal circumstances throughout each individual algorithm model at first, as we want to determine whether the algorithms are correct at all. Analysis involving unreliable communication and modifying the network by adding or removing nodes will be covered in Chapter 10.

As mCRL2 has difficulty analyzing timed models, we present two different models in this section: the first model is constructed in mCRL2 whereas the second model is constructed using UPPAAL.

6.1 mCRL2 model

Naturally, we need a process for each SmartPixel - and since all SmartPixels have identical behavior, we just use the same process multiple times in parallel. We introduce actions `sendID`, `recvID` and `commID` with parameters *num,id* - these actions are used to respectively send, receive and communicate id *id* to node number *num*.

It seems we are on the right track now: each node can communicate what it believes to be the leader id to any adjacent node and this can be handled accordingly. However, one issue remains: the protocol specification states that after a few seconds, the leader election phase is done and thus a node that still considers itself to be the leader must become the leader. Unfortunately, the mCRL2 tools have difficulties analyzing models which involve timing. Thus, we try to prevent timing for now by coming up with some way to determine whether the desired result has been achieved; strictly speaking, we are checking whether the desired result *will* occur, we are not considering *when* it occurs. The latter can only be checked by analyzing the timed models, which we will attempt later on. By checking whether the desired result will occur we are actually proving correctness of the algorithm, which is desirable indeed.

Analyzing the algorithm reveals that at the end of the leader election phase, any node that has its own id as leader id will become a leader (line 18) - in other words, the node has not received an id which is lower than its own id. From this observation, it follows that if a node has ever received an id lower than its own id, it will *never* become a leader. Thus, if we introduce a monitoring process that counts the number of nodes that receive a lower leader id while they still consider themselves as the leader, we are in business: if we have n nodes in the network, the election phase ends if we have received word that $n - 1$ nodes will not become leader.

To this end, we introduce the `gotLeader` and `monLeader` actions, all with parameters *num*, *id*, *initial* which communicate together as `commLeader`: these are used to indicate that node number *num* received leader id *id*. The *initial* boolean is *true* if receiving *id* results in the node demoting itself to an ordinary node: it will no longer become leader (We note that although it may seem we provide more information to the monitor than is necessary, this approach will allow us to check whether all nodes have the same leader later on by applying only minor changes to the leader process). The effect is that per node, *initial* is *true* at most one time: a node will initially consider itself leader and can only demote itself if it believed it was a leader. A process called `LeaderMon` uses *monLeader* actions to determine the number of nodes demoting themselves to ordinary nodes, by counting the number of *monLeader* actions where *initial* = *true*. Once all but one node have declared they will not become leader, there is only a single node that tries to become leader. This is modeled using the `tryLeader` and `timeout` actions which communicate together as `leader`: thus, a `tryLeader` action can only be performed if there is a `timeout` action available. This action is performed by `LeaderMon` if all but one of the nodes have indicated they will not be a leader.

It must be noted that the above reasoning does not seem to confirm to the algorithm we are attempting to prove correct. Indeed, the algorithm as presented has absolutely no notion of how many nodes there are, let alone if they have a leader or not. Fortunately, this does not matter: we intend to show that always one node elects itself as leader. In our model, the final node only becomes a leader if all other nodes do not elect themselves as leader, which is essentially the same as waiting a fair period of time before concluding that since there apparently is no one else available, this node must be the leader.

Summarizing, we have the following actions:

- `gotLeader(num,leaderid,initial)` | `monLeader(num,leaderid,initial)` → `commLeader(num,leaderid,initial)`
  The pair of `gotLeader` and `monLeader` actions are used to indicate that node number *num* believes *leaderid* to be the leader. *initial* is true the first time the node has decided it will not become leader.
- `sendID(num,id)` | `recvID(num,id)` → `commID(num,id)`
  These actions are used to send leader id *id* to node number *num*.
- `tryLeader(n)` | `timeout(n)` → `leader(n)`
  The `tryLeader` action is issued by a node if it attempts to become leader; it is used in conjunction with `timeout`, which is generated by the `LeaderMon` process. The *n* is the id of the node that tries to become leader.

We shall now illustrate the main `Smartpixel1` process:

```
35 proc        Smartpixel1init(num:Nat) =
36          Smartpixel1(num,mapNodeId(num),0);
37
38         Smartpixel1(num,leader_id,s:Nat) =
```

Line 35 defines the `Smartpixel1init` process - basically, this process becomes a `Smartpixel1` process for a given node number *num*. The purpose of this process is to keep the initialization compact by providing defaults: each node considers itself as leader and has not yet sent any messages. Line 38 defines the `Smartpixel1` process. The parameters are the node number *num* and the leader id *leader_id*, plus a variable *s* containing the next direction we need to send our leader id to.

```
39 (s<4&&(neighbornum(num,rr(s,RotationList.num))==num)) ->
40     intern . Smartpixel1(num,leader_id,s+1) +
41 (s<4&&(neighbornum(num,rr(s,RotationList.num))!=num)) ->
```

```
42      sendID(neighbornum(num,rr(s,RotationList.num)),leader_id) .
43       Smartpixel1(num,leader_id,s+1) +
```

If the leader id has not been communicated to a neighbor in direction s, we use the `sendID` action as illustrated above to send the notification to the specified neighbor node. As `commID` is a communication action, it is only permitted if there is a corresponding `recvID` with the same arguments, so if there is no such neighbor we perform an `intern` action. This corresponds to line 5 - 8 of the algorithm.

```
44  sum rid:Nat . (rid<leader_id) -> recvID(num,rid) .
45                                   gotLeader(num,rid,mapNodeId(num)==leader_id) .
46                                   Smartpixel1(num,rid,0) +
```

As the algorithm dictates, if we receive any leader id rid that is lower than what we currently consider as our leader ($rid < leader\_id$), we claim we have a leader by issuing the $gotLeader$ action where $initial = true$ if we considered ourself to be the leader ($mapNodeId(num) = leader\_id$) and $false$ otherwise. We finally update our process state by using rid instead of $leader\_id$ as the leader id parameter, and $s = 0$ since we need to send this new leader id to our neighbors, like the algorithm requires (lines 9 - 15)

```
47  sum rid:Nat . (rid>=leader_id) -> recvID(num,rid) .
48                                    Smartpixel1(num,leader_id,s) +
```

If we receive the same leader or a leader that has a higher id than what we currently consider as the leader ($rid \geq leader\_id$), we do not update our internal state at all (lines 16 - 17).

```
49  (mapNodeId(num)==leader_id) -> tryLeader(mapNodeId(num));
```

If we believe we are the leader, we can perform a `tryLeader` action; this is only possible if there is a subsequent `timeout` action available since these communicate. This is outlined in line 18 of the algorithm.

Where are the subsequent `timeout` actions generated? These are performed by a monitoring process `LeaderMon`. The idea is that this process collects the number of `gotLeader` actions with $initial = true$, these are generated if a node decides it will not become a leader, which happens at most a single time per process, as previously argued. Since there are n nodes in total, the `LeaderMon` process should wait for $n-1$ `gotLeader` actions before it may perform `timeout` actions, which unblock `tryLeader` actions and thus allowing nodes to assume the role of leadership. As `gotLeader` and `monLeader` communicate together, the `LeaderMon` process should simply perform `monLeader` actions:

```
50  LeaderMon(n:Int) =
51   sum a,b:Nat . monLeader(a,b,true) . LeaderMon(n-1) +
52   sum a,b:Nat . monLeader(a,b,false) . LeaderMon(n) +
```

The first line simply defines the `LeaderMon` process. The next two lines try to perform `monLeader` actions, which are only possible if there is a corresponding `gotLeader` action. If a node tells us it won't become leader (that is, the $initial$ parameter is $true$), we decrease the number of processes we are waiting for, since such an action will only happen a single time. If a `gotLeader` action is performed by a process that already gave up its leadership role (the $initial$ parameter is $false$), we do not alter the number of processes and continue.

```
53   sum a:Nat . (n<=0) -> timeout(a).LeaderMon(n);
```

Finally, if we are no longer waiting for processes, we perform `timeout` actions with parameter a whenever possible, for any a as we want to allow any remaining node id to become leader. These will cause the remaining processes that consider themselves as leader to be able to issue an `tryLeader` action, which will become visible using the communication variant `leader`.

It remains to specify the initializing process:

```
54 proc    System =
55          allow(commID,leader,commLeader,timeout, (
56           comm(sendID|recvID->commID,tryLeader|timeout->leader,
57            gotLeader|monLeader->commLeader, (
58             Smartpixel1init( 0) || Smartpixel1init( 1) || Smartpixel1init( 2) ||
59             Smartpixel1init( 3) || Smartpixel1init( 4) || Smartpixel1init( 5) ||
60             Smartpixel1init( 6) || Smartpixel1init( 7) || Smartpixel1init( 8) ||
61             LeaderMon(DIM * DIM - 1)
62            ))
63           ));
64
65 init    System;
```

Line 55 specifies the actions we allow. We use this to block the explicit `sendID`, `recvID`, `gotLeader`, `monLeader`, `tryLeader` and `timeout` actions, as we are only interested in their communication variants. The next line specifies the communication between actions, which we have outlined previously. Finally, we list the processes: this is a 3×3 grid, totaling 9 nodes. Thus, our monitor has to observe all but one node deciding they will not become leader before we unblock the `leader` action.

6.2 mCRL2 results

As outlined previously, we want to show two different properties: only one leader gets elected, and all nodes agree on the same leader. We introduce two properties, which we prove by inspecting the mCRL2 models.

Property 6.1. *The algorithm LeaderElection as modeled in Section 6.1 always causes exactly one node to consider itself as the leader*

Proof. We generate a labeled transition system of the model, where we hide everything except the *leader* actions. Generating the state space using rooted branching bisimilarity (which is the equivalence relation we shall use from now on) of a 3×3 configuration yields Figure 6.1.

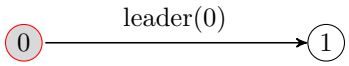


Figure 6.1: Leader election for Property 6.1, in which only the *leader* action is visible

We observe that there is only one path that always results in node 0 becoming the leader, as claimed. □

Property 6.1 can also be expressed using two modal formulas (we disregard the parameters of the *leader* action here, as these are not important):

- $[true^\star \cdot leader \cdot true^\star \cdot leader]false$
  This expresses it cannot happen that two consecutive `leader` actions occur.

- $[\overline{leader}^{\star}]\langle true^{\star} \cdot leader\rangle true$
  If a `leader` action has not yet occured, it will eventually occur.

As expected, checking these formulas in the original model without hiding any actions results in all of them holding. We need the second formula to show that the `leader` action indeed occurs in the model - if this was not the case, the first formula would always be true.

This leads us to prove our second statement:

Property 6.2. *The algorithm LeaderElection as modeled in Section 6.1 results in all nodes eventually ending up in a state where they agree upon the same leader.*

Proof. In our current model, a node will perform a `gotLeader(id,rid,id==leader_id)` action once it receives a lower leader id *rid* (refer to line 45 of the algorithm) - this action is then used to unblock nodes from assuming leadership. However, we can also replace this action by `gotLeader(id,rid,rid==0)`, which means the `timeout` will be blocked until all non-leader nodes believe that node 0 is their leader. The resulting state space is shown in Figure 6.2.

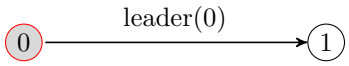


Figure 6.2: Leader election for Property 6.2 in which only the *leader* action is visible

We observe that the `leader` action will always be performed, which means all non-leader nodes consider node 0 as their leader, which is what we claimed. □

If we were to use a modal formula, we need to check that $[\overline{leader}^{\star}]\langle true^{\star} \cdot leader\rangle true$ holds, which is indeed the case.

6.3 UPPAAL model

As mentioned previously, mCRL2's current toolkit is not capable of analyzing timed models. Thus, we introduce a model in the UPPAAL modeling toolset [BLL+95]. In UPPAAL, systems are described by networks of timed automata. We will not discuss the precise definition of a timed automaton here, but rather illustrate the overall idea: an automaton consists of a set of states (visualized by circles) and transitions between those states (visualized by arrows). The first state the system starts in is called the *initial state*, which is visualized by a double circle.

States can have *invariants* attached to them: these are conditions that must hold as long as the system resides in that state. Consequently, if the invariant condition no longer holds, a transition to another state must be taken. If a state contains a condition, it is placed within the circle below the horizontal line.

Transitions can have up to four different attributes, all of which are optional. There may be an *update*, consisting of a *variable* ← *expression* combination: once the transition has been taken, the contents of *variable* is set to *expression*. There may also be a *guard*, which is a predicate that must be true for this transition to be taken. For example, if the condition is $z < 5$, the transition may only be taken if the value of z is smaller than 5. Another possible attribute is a *selection* value, which represents choice of a value out of some domain: for example, if we have $b : Bool$, this indicates variable b can assume any value from the *Bool* domain, so in this case, $b = true$ or $b = false$. This b can subsequently be used in other attributes. Finally, there may be *synchronization*, which indicates interaction between multiple processes. There are two flavors: *comm*! and *comm*? - the first one indicates a *comm* communication is outgoing, whereas the latter indicates a *comm* communication is incoming. For example, if one process

has a transition which contains *comm*! and all other attributes are satisfied, another process that has a transition that contains *comm*? and also having all other attributes satisfied will allow both transitions to be taken; this makes it possible to model communication between multiple processes. Synchronizations can have parameters as well: if some process contains a transition with $comm[true][false]$! and another process has selection $b, c : Bool$ and $comm[b][c]$? could cause *both* transitions, the sending process as well as the receiving process, to be taken where $b = true$ and $c = false$. If there is a *comm*! communication without a *comm*? communication in another process, the process issuing the *comm*! is not allowed to perform the transition.

In Figure 6.3, a SmartPixel performing leader election is illustrated, where all conditions are always in a fixed order: selection, guard, synchronization and update. For each SmartPixel, *id* is both the unique id associated with the SmartPixel as well as the number in the grid, *lid* is the id which the pixel considers to be the leader (initially, a node considers itself to be the leader due to the $lid \leftarrow id$ assignment) and y and z are clock variables: these are initially zero, and will increase as time passes. However, they can be reset back to zero; this makes it convenient to model time relative to some event. The $sendID[a][id]$ communication represents sending and receiving id a to node number *id*. As in the mCRL2 case, we introduce $nn(id, s)$, for which it holds that $nn(id, s) = neighbornum(id, s)$. Finally, we have a type $NodeSet$, which contains all possible node id's; for example, in a 3×3 network, $NodeSet = \{0, \ldots, 8\}$. The complete model is illustrated in Figure 6.3.

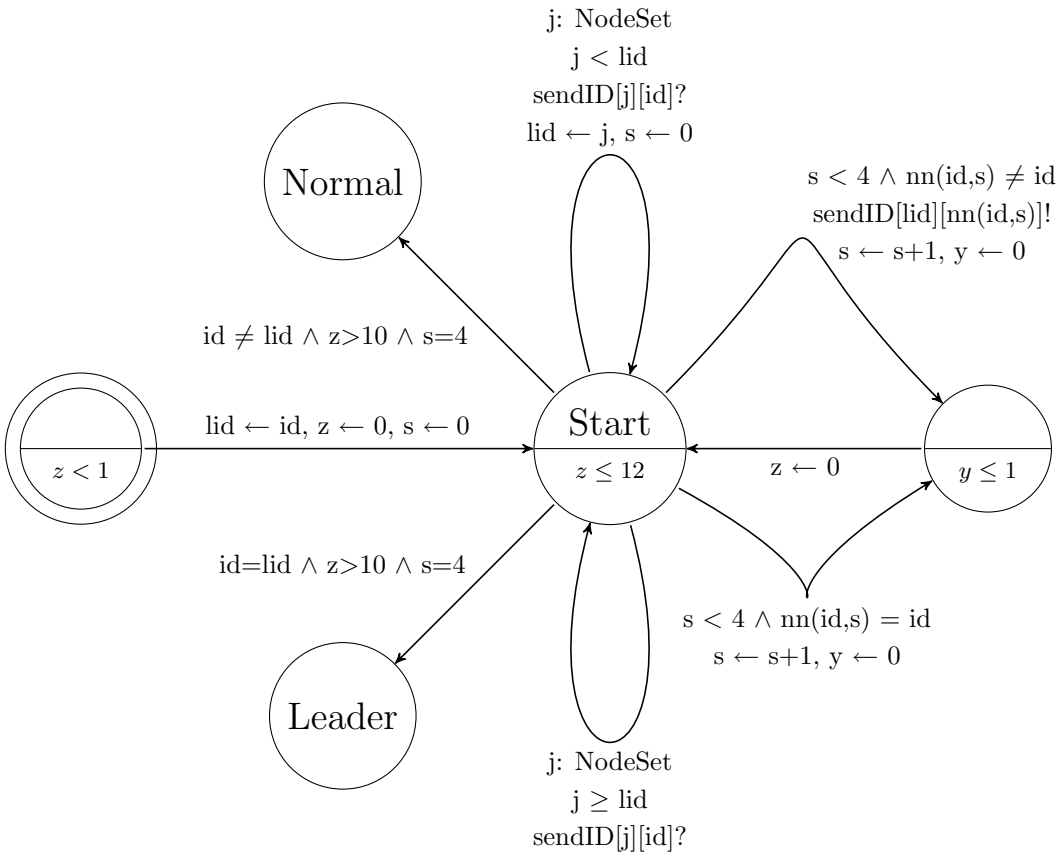


Figure 6.3: Leader election model in UPPAAL

6.4 UPPAAL results

As previously outlined, UPPAAL allows us to formulate requirements as a series of temporal logic formula's; the logic used is a subset of computational tree logic [HR04]. The major difference is that modal operators may not be nested: each formula must contain only a single modal operator with either a $\Diamond$ or a $\Box$, the remainder of the formula cannot have any modal operators. To this end, we have formulated the following formula's:

1. $A\Diamond\forall_{i:NodeSet}(Node(i).Leader \vee Node(i).Normal)$
   A state in which all nodes are in the *Leader* or *Normal* state is eventually reachable.
2. $A\Box\forall_{i:NodeSet}\forall_{j:NodeSet}(Node(i).Leader \wedge Node(j).Leader \Rightarrow i = j)$
   There is never more than a single node in the *Leader* state.
3. $A\Diamond\exists_{i:NodeSet}Node(i).Leader$
   Eventually, there will be at least a single node in *Leader* state.
4. $A\Diamond\forall_{i:NodeSet}Node(i).lid = 0$
   All nodes end up in a state where they consider node id 0 to be their leader.

We note that items 2 and 3 together imply that eventually, there has to be exactly one leader. Combined with Property 6.1, the system has to end in a state where all but one nodes are in the *Normal* state and a single node is in the *Leader* state. Finally, item 4 insists that all nodes agree on the same leader.

Asking UPPAAL to perform verification of these formulas yields the conclusion that all these properties are satisfied.

Chapter 7

Node coordinates determination

7.1 The model

The immediate goal of this phase is clear: we want to show, that the *DetermineCoords* algorithm results in all nodes receiving correct coordinates. To fulfill this goal, we intend to have all nodes report the coordinates they obtain to a monitor, which in turn counts the number of correct coordinates received. However, in order to do this, we must first define what the correct coordinates are. Let us consider how coordinates are defined: basically, this is the position of the node relative to the leader - in other words, the leader itself always resides at coordinates $\langle 0, 0 \rangle$. Thus, as illustrated in Figure 4.5, if a node at direction 0 exists, it should obtain coordinates $\langle 0, 1 \rangle$. From this observation, it follows that the desired coordinates of some node number n is simply the difference between node n's position and the position of the leader (note that we do not consider rotations here - this is not necessary, since all nodes assume the rotation of the leader once they have received their coordinates). However, there is an important observation to be made: as outlined in Chapter 5, in the model nodes are numbered sequentially from $0 \ldots N-1$, where node 0 is the top-left node and node $N-1$ is the bottom-right node. Yet, our coordinate system as described in Figure 5.1 considers bottom-left as the smallest (x, y)-coordinates and top-right the largest. Due to this distinction between the model and our coordinate system, we have to invert the y-coordinate. To this end, we define a function $calcCoord(n) = \langle n \bmod \mathcal{D} - \mathcal{L} \bmod \mathcal{D}, \mathcal{L} \operatorname{div} \mathcal{D} - n \operatorname{div} \mathcal{D} \rangle$ which calculates the coordinates of node number n. In mCRL2, this becomes:

```
66  map     calcCoord : Int -> Coord;
67  var     num:Int;
68  eqn     calcCoord(num) =
69                    coord(num mod DIM - LEADERNUM mod DIM,
70                          LEADERNUM div DIM - num div DIM);
```

The first two lines are simply administration for defining the function and a dummy argument. The remaining lines define the $calcCoord(num)$ function as stated before.

We need to introduce actions in order to send and receive messages, as well as a communication action to connect these. To this end, consider the situation where node A sends a message to node B: we introduce the actions `sendCoord`, `recvCoord` and `commCoord`, all with parameters $num, coord, wantdir, recvdir$, where num is the number of the node B, the receiving node (this is outlined in Chapter 5), $coord$ are the coordinates which the sending node A transmitted, $wantdir$ is the direction in which node B should receive the message, whereas direction $recvdir$ is the direction in which node B actually received the message. We define `sendCoord` | `recvCoord` → `commCoord` in order to model communication between the processes.

There is a crucial piece of information that our model requires: if we are to issue a *recvCoord* action, we must limit the direction from which the message can be received, since there is only one possible direction in which such a message can be received. How we can determine this direction? The intuition is that the node that sends the message will already have assumed the rotation of the leader. This argument holds by induction: initially, the leader issues all coordinates so the argument trivially holds. As for the induction case, the node initiating the communication already has the correct orientation, or it would not be initiating communication.

In order to determine the receiving direction d' of a message given that it should arrive on direction d on node *num* with rotation r, we observe this is the difference between the node's rotation r and the rotation of the leader node. However, the rotation of the leader is always zero: this means that the receiving direction is always relative to no rotation, from which it follows that the direction d' on which the message arrives is simply direction d left-rotated r times: the node is right-rotated r times, so we need to left-rotate it r times in order to counter this. This means that $d' = rl(d, RotationList.num)$.

While inspecting the algorithm, we notice that once coordinates are received, the node will consider itself to be at these coordinates. Subsequently, receiving any other coordinates will result in `reset` actions, as this could only happen if there are multiple leaders in the system. Once all nodes except the leader have received coordinates, we are done. To this end, we introduce the `success` action in our model, along with actions to report and acknowledge coordinates `reportCoord`, `ackCoord`, all with parameters *num*, *coord*, which communicate together as `reportCoord | ackCoord →` `coordAction`. Once a node receives its coordinates, it will initiate the `reportCoord` action, where *num* is the node number and *coord* are the coordinates it considers itself at. As before, we use a `ResultMon` process which acknowledges `reportCoord` actions and keeps track of how many correct coordinates have been reported. Once all nodes have reported correct coordinates, a `success` action will be initiated.

Summarizing, we have the following actions:

- `sendCoord(num,coord,wantdir,recvdir) | recvCoord(num,coord,wantdir,recvdir)` `→ commCoord(num,coord,wantdir,recvdir)`
  These actions are used to subsequently send, receive and communicate coordinates to a neighboring node.
- `reportCoord(num,coord) | ackCoord(num,coord) → coordAction(num,coord)`
  Once node number *num* receives coordinates *coord*, it performs `reportCoord` action with said parameters. The monitor process uses `ackCoord` actions to verify these coordinates.
- `success`
  If all nodes but one (the leader, as it always resides on fixed coordinates) have reported their coordinates, the result monitor process will initiate a `success` action if all coordinates were correct.
- `reset`
  If a node has already received coordinates, yet receives new coordinates which do not agree on the previous ones, a `reset` action must be initiated.

We will now illustrate the `Smartpixel2` process:

```
71 proc    Smartpixel2init(num:Nat) =
72          Smartpixel2(num,num==LEADERNUM,if(num==LEADERNUM,0,4),coord(0,0),0);
73
74         Smartpixel2(num:Nat,actAsLeader:Bool,s:Nat,log:Coord,r:Int) =
```

We introduce the `Smartpixel2init` process, which is convenient for initializing the main `Smartpixel2` process using default parameters. The node assumes its initial coordinates are $\langle 0, 0 \rangle$, but only the

leader will initialize sending coordinates to neighboring nodes. Finally, the rotation is initially 0, causing nodes to assume they have not been rotated.

```
75 (s<4&&neighbornum(num,rr(s,RotationList.num))==num) -> intern .
76           Smartpixel2(num,actAsLeader,s+1,log,r) +
77 sum d':Direction.(s<4&&neighbornum(num,rr(s,RotationList.num))!=num) ->
78          sendCoord(neighbornum(num,rr(s,RotationList.num)),move(rr(s,r),log),
79           rr(rd(s),r),d') .
80          Smartpixel2(num,actAsLeader,s+1,log,r) +
```

As illustrated in Section 5, we calculate the neighbor node number using the *neighbornum* and *rr* functions. We then proceed to calculate the coordinate to send and the direction we want it to arrive in. The reason we need a sum over the receiving direction, is that the sending node cannot know at which direction such a message arrives - thus, we try to send the message to any receiving direction possible. The corresponding *recvCoord* function will then in turn only accept a single receiving direction. This corresponds to lines 7 - 11 of the algorithm.

```
81 sum c:Coord,d,d':Direction.(log==coord(0,0)&&!actAsLeader&&d'==rl(d,RotationList.num)) ->
82        recvCoord(num,c,d,d') . reportCoord(num,rl(d,d'),c) .
83        Smartpixel2(num,actAsLeader,0,c,rl(d,d')) +
```

If coordinates are received while we still consider us at position $\langle 0, 0\rangle$ and we do not act as leader, we honor these coordinates and report them accordingly to the monitor process. We set s to 0, since we should inform our neighbors of their coordinates relative to our own. This corresponds to lines 14 - 18 of the algorithm.

```
84 sum c:Coord,d,d':Direction.(c!=log&&(log!=coord(0,0)||actAsLeader)) ->
85        recvCoord(num,c,d,d') . reset +
```

Once a node receives new coordinates ($c \neq log$) while it already knows its coordinates ($log \neq \langle 0, 0\rangle \vee actAsLeader$), any different coordinates must be rejected using initiation of the `reset` action (line 22 of the algorithm).

```
86 sum d,d':Direction . ((log!=coord(0,0)||actAsLeader)&&d'==rl(d,RotationList.num)&&
87       r==rl(d,d')) ->
88        recvCoord(num,log,d,d') . Smartpixel2(num,actAsLeader,s,log,r) +
```

If a node receives identical coordinates and the rotation matches, the status is unchanged (line 28 of the algorithm).

```
89 sum d,d':Direction . ((log!=coord(0,0)||actAsLeader)&&d'==rl(d,RotationList.num)&&
90       r!=rl(d,d')) ->
91        recvCoord(num,log,d,d') . reset;
```

If a node receives identical coordinates but the rotation does not agree with what the node previously determined, a `reset` action must be initiated (line 24 of the algorithm).

We still need to specify the `ResultMon` process, which uses `ackCoord` actions to monitor the result of `reportCoord` actions:

```
92 proc    ResultMon(ok:Int) =
93          sum num:Nat.ackCoord(num,RotationList.num,calcCoord(num)) . ResultMon(ok+1) +
94          sum num,r:Nat,c:Coord. (c!=calcCoord(num)||r!=RotationList.num) ->
95           ackCoord(num,r,c) . ResultMon(ok) +
```

The first line defines the `ResultMon` process. There is a single parameters: `ok` indicates how many valid node coordinates have been received. This is illustrated in the next two lines of the model: only if the position for node number *num* is identical to *calcCoord*(*num*) and the rotation matches the item in the rotation list, we increase *ok* by one. Otherwise, we acknowledge the coordinate but do not change our status.

```
96  (ok>=(DIM*DIM-1)) -> success;
```

Finally, if all nodes except the leader have reported their coordinates, we are done - this implies the `success` action will only be initiated if all nodes have successfully received their coordinates, which is what we wanted to determine.

It remains to specify the initializing process:

```
97  proc    System =
98           allow(commCoord,coordAction,success,reset, (
99            comm(sendCoord|recvCoord->commCoord,reportCoord|ackCoord->coordAction, (
100              Smartpixel2init( 0) || Smartpixel2init( 1) || Smartpixel2init( 2) ||
101              Smartpixel2init( 3) || Smartpixel2init( 4) || Smartpixel2init( 5) ||
102              Smartpixel2init( 6) || Smartpixel2init( 7) || Smartpixel2init( 8) ||
103              ResultMon(0,0)
104           ))
105          ));
106
107 init    System;
```

Line 98 specifies the actions we allow. We use this to block the explicit `sendCoord`, `recvCoord`, `reportCoord` and `ackCoord` actions, as we are only interested in their communication variants. The next line specifies the communication between actions, which we have outlined previously. Finally, we list the processes: this is a 3×3 grid, totaling 9 nodes. We initialize our monitor process, which hasn't received any reports yet, so the parameter is zero.

7.2 The results

First of all, what do we intend to show? The goal is, that whenever *DetermineCoords* ends, the coordinates of node number *num* confirm to the output of the function *calcCoord*(*num*). This property can be proved by inspecting the mCRL2 model.

Property 7.1. *The algorithm DetermineCoords as modeled in Section 7.1 results in all nodes assuming coordinates calcCoord*(*num*)*, where num is the subsequent number of the node in question.*

Proof. We note that line 93 of our model registers the number of correct positions reported. Subsequently, line 96 initiates a `success` action once all correct positions have been acknowledged. To this end, we generate a labeled transition system in which we only show the `success` and `reset` actions. The result is illustrated in Figure 7.1.

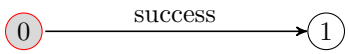


Figure 7.1: Grid determination in which only the `success` and `reset` actions are visible

We observe that the `success` action will always occur, which proves the algorithm *DetermineCoords* as modeled in Section 7.1 results in the correct coordinates being assigned, as claimed. □

Property 7.1 can also be expressed as a modal formula, which is $[\overline{success}^\star]\langle true^\star \cdot success\rangle true$: as long as a *success* has not been performed, it must eventually be possible to do so. As expected, this property holds in the model.

Chapter 8

Grid size determination

8.1 The model

In order to check whether the algorithm determines the correct grid size, we need to introduce two functions: $makeMinCoords$ and $makeMaxCoords$ which determine the corresponding minimum and maximum coordinates of a $\mathcal{D} \times \mathcal{D}$ grid with leader number $\mathcal{L}$. Let us consider Figure 8.1: we observe the minimum position is the bottom-left coordinate, $\langle -2, -1 \rangle$, whereas the maximum position is the top-right coordinate, $\langle 0, 1 \rangle$. The question is: how can we determine the minimum and maximum positions if we only know the grid dimensions and the leader node number? If we study the figure in more detail, we observe that the minimum position consists of the amount of nodes to the bottom-left of the leader node: there are two nodes to the left of the leader node and one node below the leader node, so the minimum position is $\langle -2, -1 \rangle$. Likewise, the maximum position consists of the number of nodes to the top-right of the leader node: there is one node above the leader node, and zero nodes to the right of the leader node, so the maximum position is $\langle 0, 1 \rangle$.

0 $\langle -2, 1 \rangle$	1 $\langle -1, 1 \rangle$	2 $\langle 0, 1 \rangle$
3 $\langle -2, 0 \rangle$	4 $\langle -1, 0 \rangle$	5 $\langle 0, 0 \rangle$
6 $\langle -2, -1 \rangle$	7 $\langle -1, -1 \rangle$	8 $\langle 0, -1 \rangle$

Figure 8.1: Sample setup of a grid where $\mathcal{D} = 3$ and $\mathcal{L} = 5$

Since we know the dimensions of the grid and the nodes are numbered sequentially, we can easily determine the amount of nodes to the left of our leader: this is $\mathcal{L} \bmod \mathcal{D}$. The amount of nodes above our leader is $\mathcal{L} \operatorname{div} \mathcal{D}$. Since grid x-coordinates are from left to right, the leftmost coordinate is the smallest, so the minimum x-coordinate is $-(\mathcal{L} \bmod \mathcal{D})$. The grid y-coordinates are from bottom to top, which means that we need to determine the number of nodes above the leader node. This means the minimum y-coordinate is $-((\mathcal{D} - 1) - (\mathcal{L} \operatorname{div} \mathcal{D}))$.

Determining the maximum coordinates of the grid can be performed in a similar session: as stated, we need to determine the amount of nodes to the right of the leader node. Since there are

$\mathcal{L} \bmod \mathcal{D}$ nodes to the left of the leader node, there are $(\mathcal{D}-1)-(\mathcal{L} \bmod \mathcal{D})$ nodes to the right of the leader node, so the maximum x-coordinate is $(\mathcal{D}-1)-(\mathcal{L} \bmod \mathcal{D})$. Finally, we need to determine the number of nodes above the leader node: this is $\mathcal{L} \operatorname{div} \mathcal{D}$, which means the maximum y-coordinate is $\mathcal{L} \operatorname{div} \mathcal{D}$. Summarizing, we can define $makeMinCoords = \langle -a, -((\mathcal{D}-1)-b)\rangle$ and $makeMaxCoords = \langle (\mathcal{D}-1)-a, b\rangle$ where $a = \mathcal{L} \bmod \mathcal{D}$ and $b = \mathcal{L} \operatorname{div} \mathcal{D}$.

As usual, we need to introduce actions in order to model the communications. To this end, we introduce actions `sendMinMax`, `recvMinMax` and `commMinMax`, all with parameters $num, leaderid, min, max$: current leader id $leaderid$ and min, max which are considered to be the minimum and maximum coordinates are communicated with node number num. Again, `commMinMax` is the communication function between `sendMinMax` and `recvMinMax`.

Once a node retrieves updated min/max coordinates, we need some way to verify these. This is why we define the `reportMinMax`, `checkMinMax` and `commMinMax2` actions, all using parameters num, min, max, stating node number num has reported new minimum/maximum positions min/max. A monitor process uses `checkMinMax` actions to validate the positions: whenever a node reports mix/max coordinates, we acknowledge them and keep track of the number of successful or invalid reports.

In order to be able to report that the desired state has been reached, we introduce `success` and `reset` actions as usual - the monitor process will issue a `success` once all nodes have reported correct minimum/maximum coordinates.

Summarizing, we have the following actions:

- `sendMinMax(num,leaderid,min,max)` | `recvMinMax(num,leaderid,min,max)`
  → `commMinMax(num,leaderid,min,max)`
  These actions are used to send, receive and communicate leader id $leaderid$ and minimum/maximum coordinates min/max to node number num.
- `reportMinMax(num,min,max)` | `checkMinMax(num,min,max)`
  → `commMinMax2(num,min,max)`
  Once node number num retrieves new min/max coordinates min/max, it will use a `reportMinMax` action to inform a monitor process, which uses the `checkMinMax` action to verify the coordinates.
- `success`
  Once all nodes have reported their min/max coordinates, a `success` action is issued if all coordinates are correct.
- `reset`
  Whenever contradictory messages are being received, the node must be reset.

Before we discuss the main process, we need to declare functions that return the minimum and maximum of two coordinates, as outlined in Section 4.3. We call these functions $minCoord$ and $maxCoord$ and give their corresponding declarations immediately:

```
map     maxCoord : Coord#Coord -> Coord;
        minCoord : Coord#Coord -> Coord;
var     x1,y1,x2,y2: Int;
eqn     maxCoord(coord(x1,y1),coord(x2,y2)) = coord(max(x1,x2),max(y1,y2));
        minCoord(coord(x1,y1),coord(x2,y2)) = coord(min(x1,x2),min(y1,y2));
```

We will now introduce the model:

```
proc    Smartpixel3init(num:Nat) =
         Smartpixel3(num,num==LEADERNUM,mapNodeId(LEADERNUM),if(num==LEADERNUM,0,4),
                     calcCoord(num),calcCoord(num),calcCoord(num));
```

```
116
117         Smartpixel3(num:Nat,actAsLeader:Bool,lid,s:Nat,log,mincoord,maxcoord:Coord) =
```

We introduce the `Smartpixel3init(num)` process, which is used for initialization of a leader or ordinary node based on the node number *num*. Initial default parameters are given to keep the initialization of the model more compact. We then introduce the `Smartpixel3` process itself.

```
118 (s<4&&neighbornum(num,rr(s,RotationList.num))==num) -> intern .
119     Smartpixel3(num,actAsLeader,lid,s+1,log,mincoord,maxcoord) +
120 (s<4&&neighbornum(num,rr(s,RotationList.num))!=num) ->
121     sendMinMax(neighbornum(num,rr(s,RotationList.num)),lid,mincoord,maxcoord) .
122     Smartpixel3(num,actAsLeader,lid,s+1,log,mincoord,maxcoord) +
```

If a node has not sent its min/max coordinates to direction s, let it do so and the update s. This corresponds with lines 7 - 10 of the algorithm.

```
123 sum n:Nat,cmin,cmax:Coord . (n>lid) -> recvMinMax(num,n,cmin,cmax) . reset +
```

If min/max coordinates are received from a higher leader id, it means there must be multiple leaders and we reset the system (lines 13 - 15).

```
124 sum n:Nat,cmin,cmax:Coord . (n<=lid&&
125     (minCoord(cmin,mincoord)!=mincoord||maxCoord(cmax,maxcoord)!=maxcoord)) ->
126      recvMinMax(num,n,cmin,cmax) .
127      reportMinMax(num,minCoord(cmin,mincoord),maxCoord(cmax,maxcoord)) .
128      Smartpixel3(num,actAsLeader,n,0,log,
129       minCoord(cmin,mincoord),maxCoord(cmax,maxcoord)) +
```

If smaller minimum or greater maximum coordinates have been received, report them to the monitor process and update the process accordingly (algorithm lines 17 - 24)

```
130 sum n:Nat,cmin,cmax:Coord . (n<=lid&&minCoord(cmin,mincoord)==mincoord&&
131     maxCoord(cmax,maxcoord)==maxcoord) ->
132        recvMinMax(num,n,cmin,cmax) .
133        Smartpixel3(num,actAsLeader,n,log,mincoord,maxcoord,s);
```

If coordinates are received which do not influence the min/max coordinates we currently have, we only need to honor the leader id, which can only be smaller or equal than what we currently know (algorithm line 27)

```
134 proc    ResultMon(ok:Int) =
135          sum id:Pos,cmin,cmax:Coord . ((x(cmax)-x(cmin)+1 == DIM) &&
136                                        (y(cmax)-y(cmin)+1 == DIM) &&
137                                        ((cmin!=makeMinCoords)||(cmax!=makeMaxCoords))) ->
138                                       checkMinMax(id,cmin,cmax) .  ResultMon(ok) +
```

This is the `ResultMon` process, with a single parameter *ok*. This parameter stores the number of correct min/max coordinates that have been received. The next line checks once the desired grid size of $\mathcal{D} \times \mathcal{D}$ is reported, whether the expected min/max coordinates have been received. If this is not the case, the status is not updated.

```
139 sum id:Pos,cmin,cmax:Coord . ((x(cmax)-x(cmin)+1 == DIM) &&
140                              (y(cmax)-y(cmin)+1 == DIM) &&
```

```
141                                (cmin==makeMinCoords&&cmax==makeMaxCoords)) ->
142                               checkMinMax(id,cmin,cmax) .  ResultMon(ok+1) +
```

If a grid size is reported with min/max coordinates as expected and the dimensions are correct, we have one more node storing the correct min/max, so we update our state accordingly.

```
143 sum id:Pos,cmin,cmax:Coord . ((x(cmax)-x(cmin)+1 <  DIM) ||
144                               (y(cmax)-y(cmin)+1 <  DIM)) ->
145                              checkMinMax(id,cmin,cmax) . ResultMon(ok) +
146 sum id:Pos,cmin,cmax:Coord . ((x(cmax)-x(cmin)+1 >  DIM) ||
147                               (y(cmax)-y(cmin)+1 >  DIM)) ->
148                              checkMinMax(id,cmin,cmax) . ResultMon(ok) +
149 (ok>=DIM*DIM) -> success;
```

Grid sizes that are smaller than the final dimensions can be ignored, as the nodes are probably still working out how large the grid is. However, it is obviously not correct if bigger dimensions are being reported than we actually expect, so do not register success in such a case. Finally, if all nodes have reported the grid size correctly, we initiate a `success` action.

```
150 proc    System =
151          allow(commMinMax,commMinMax2,success,reset, (
152           comm(sendMinMax|recvMinMax->commMinMax,reportMinMax|checkMinMax->commMinMax2, (
153              Smartpixel3init( 1) || Smartpixel3init( 2) || Smartpixel3init( 3) ||
154              Smartpixel3init( 4) || Smartpixel3init( 5) || Smartpixel3init( 6) ||
155              Smartpixel3init( 7) || Smartpixel3init( 8) || Smartpixel3init( 9) ||
156              ResultMon(0)
157           ))
158          ));
```

We begin by specifying the actions we allow. Again, this is used to block explicit `sendMinMax`, `recvMinMax`, `reportMinMax` and `checkMinMax` actions, since we are only interested in the communication between actions. Finally, we list the processes themselves.

8.2 The results

The goal of this phase is to determine whether all nodes receive the correct minimum and maximum grid coordinates, where the correct minimum coordinate is $makeMinCoords$ and the correct maximum coordinate is $makeMaxCoords$ as previously described.

Property 8.1. *The algorithm DetermineGridSize as modeled in Section 8.1 results in all nodes considering $makeMinCoords$ as minimum grid coordinates and $makeMaxCoords$ as maximum grid coordinates.*

Proof. We note that line 142 of the model increases the count of correct coordinates reported if the dimensions are correct and the coordinates reported are correct. Consecutively, line 149 reports success if and only if *all* nodes have reported the correct position. Thus, we generate a labeled transition system in which we only show the `success` and `reset` actions. The result is illustrated in Figure 8.2.

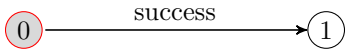


Figure 8.2: Grid size determination in which only *success* and *reset* actions are visible

We observe that the `success` action will always be performed, whereas the `reset` action will not. We observe that all nodes will end up in a state where they know the minimum/maximum grid coordinates as defined by $calcMinCoord$ and $calcMaxCoord$ as claimed. □

Property 8.1 can also be expressed as a modal formula, which is $[\overline{success}^\star]\langle true^\star \cdot success\rangle true$: as long as a `success` has not been performed, it must eventually be possible to do so. As expected, this property holds in the model.

Chapter 9

Showtime

9.1 The model

The desired goal of this phase is pretty obvious; the resulting system should only have two phases: one in which all lights are set up to display figure 0 and one in which all lights are set up to display figure 1. In the previous section, we have declared formulas that determine whether the light should be switched on or off, so we introduce an action `updateLight` with parameters n, f which indicates that node number n updates its light to reflect the output of $light(pos_n, min_n, max_n, f)$, where pos_n is the position of node n, min_n/max_n are the minimum/maximum coordinates stored by node n - whether this actually switches the light on or off doesn't need to be considered in the model.

We also need some way to communicate new status updates, so we introduce actions `sendUpdate`, `recvUpdate` and `commUpdate`, all with parameters $num, leaderid, figure$, where num is the node number to which the update is sent, $leaderid$ is the id that the sending node considers to be the leader and $figure$ is the new figure to be displayed. As usual, `commUpdate` is a communication action between `sendUpdate` and `recvUpdate`.

The algorithm states that the leader should initiate the procedure by sending figure updates every two seconds. To this end, we introduction an action `timer` that is issued when the two seconds have passed. Finally, the algorithm is also capable of resetting the system, so we introduce a `reset` action.

Summarizing, we have the following actions:

- `sendUpdate(num,leaderid,figure)` | `recvUpdate(num,leaderid,figure)`
  → `commUpdate(num,leaderid,figure)`
  These actions are used to subsequently send, receive an communicate a figure update.
- `updateLight(n,f)`
  This action is issued if node number n updates its light to display figure f.
- `timer`
  This action can be issued every two seconds, it is used to model the two-second timer.
- `reset`
  If any node receives a message stating a leader that conflicts with its own leader, a `reset` action is initiated.

We now introduce the model:

```
159 proc    Smartpixel4init(num:Nat) =
160          updateLight(num,0) .
161          Smartpixel4(num,num==LEADERNUM,mapNodeId(LEADERNUM),0,0,calcCoord(num));
162
163         Smartpixel4(num:Nat,actAsLeader:Bool,leaderid,figure,s:Nat,log:Coord) =
```

We introduce the `Smartpixel4init(num)` process, which used to initialize a node. This is used to pass predefined parameters to the `Smartpixel4` processes in order to keep the initialization code readable. It remains to specify the main `Smartpixel4` process:

```
164 (s<4&&neighbornum(num,rr(s,RotationList.num))==num) -> intern .
165     Smartpixel4(num,actAsLeader,leaderid,figure,s+1,log) +
166 (s<4&&neighbornum(num,rr(s,RotationList.num))!=num) ->
167     sendUpdate(neighbornum(num,rr(s,RotationList.num)),
168      if(actAsLeader,mapNodeId(num),leader_id),figure) .
169     Smartpixel4(num,actAsLeader,leaderid,figure,s+1,log) +
```

If we need to send a status update in some direction, we do so - however, the leader must always use its own id while sending. This corresponds to lines 7 and 18 of the algorithm.

```
170 sum n,a:Nat . (n > leaderid) -> recvUpdate(num,n,a) . reset +
```

If a status update is received with a leader id that is higher than what we currently have, we initiate a `reset` action, as is performed by line 11 of the algorithm.

```
171 sum n,a:Nat . (n <= leaderid && a != figure) -> recvUpdate(num,n,a) .
172         updateLight(num,a) .
173         Smartpixel4(num,actAsLeader,n,a,0,log) +
```

If we receive an update with the correct leader id, we update our light status and reset all sender directions, which means we will transmit the new status to our neighboring nodes (algorithm lines 14 - 18)

```
174 sum n:Nat . (n <= leaderid) -> recvUpdate(num,n,figure) .
175         Smartpixel4(num,actAsLeader,n,figure,log,s) +
```

If we receive the current figure we are displaying with an equal or lower leader id, update our leader id and continue (algorithm line 20)

```
176 (actAsLeader) -> timer .
177            updateLight(num,(figure+1) mod 2) .
178            Smartpixel4(num,actAsLeader,leaderid,(figure+1) mod 2,0,log);
```

If we act as leader, we must initiate new state changes. This is done by updating our own initial light and figure and setting $s = 0$, so we will communicate this with our neighbors (algorithm lines 5 - 7).

```
179 proc    System =
180          allow(commUpdate,reset,updateLight,timer, (
181           comm(sendUpdate|recvUpdate->commUpdate, (
182              Smartpixel4init( 0) || Smartpixel4init( 1) || Smartpixel4init( 2) ||
183              Smartpixel4init( 3) || Smartpixel4init( 4) || Smartpixel4init( 5) ||
184              Smartpixel4init( 6) || Smartpixel4init( 7) || Smartpixel4init( 8)
185           ))
```

```
186          ));
187
188  init    System;
```

We start by specifying the allowed actions; this implies we block explicit `recvUpdate` and `sendUpdate` actions. We continue by specifying the communication actions as before and finally, we list the Smartpixel processes.

9.2 The results

The idea of the showtime phase is that all nodes end up in a state where all nodes display figure 0, followed by a timeout, followed by all nodes displaying figure 1, etc.

Property 9.1. *The algorithm Showtime as modeled in Section 9.1 results in all nodes alternating between a state in which they display figure* 0 *and a state in which they display figure* 1*; transitions between these states are made using* `timer` *actions.*

Proof. First of all, we note that by line 176, a `timer` action will cause the figure to be updated and this update communicated with neighboring nodes. Thus, what we want to show is that the system will keep performing `timer` actions: it cannot happen that this action is suddenly blocked or that the system is deadlocked. To this end, we generate a labeled transition system in which only the `timer` and `reset` actions are visible. This model is illustrated in Figure 9.1.

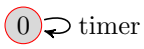


Figure 9.1: Showtime in which only the `timer` and `reset` actions are visible

We observe that `timer` actions can always be performed, as claimed. □

Property 9.1 can also be expressed using modal formula $[true^\star]\langle true^\star \cdot timer\rangle true$: in any state, it must be possible to eventually perform a `timer` action. As expected, this formula holds as this is indeed the case.

Chapter 10

Overall System

In the previous chapters, we have analyzed each phase of the protocol individually. This analysis has proved that the proposed algorithms are correct, albeit with a very strong assumption: instead of considering time, we performed an *event-based* analysis to show that the correct behavior will happen at some point in time.

While this is useful for showing algorithm correctness, it is unknown what happens if nodes are in different phases. We do expect this will become a problem in practice as some nodes may be faster than others and nodes may be added and removed in the system at any time (as outlined in the assumptions in Section 2.2). Hence, this chapter will study the system as a whole, instead of focusing on a single phase. This has the expect result of enlarging both our mCRL2 specification as well as the statespace. Since the merging of all these processes is quite straightforward, we shall only list the mCRL2 process declaration and discuss the overall steps we have performed (the complete specification is available in Appendix C):

```
189 Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr:Nat,leader,log,mincd,maxcd:Coord,f,r,rc:Nat)=
```

Where:

num	The node's internal identification number
leader_id	Identification number of the node's leader
ph	Current node phase, 0 means the node is resetting ($ph = 0 \Leftrightarrow mustReset$; refer to Section 4.5 for more information about the reset algorithm)
sl	Sent variable (s) for leader election messages
sc	Sent variable (s) for node coordinate messages
ss	Sent variable (s) for grid size messages
su	Sent variable (s) for figure update messages
sr	Sent variable (s) for reset messages
leader	The process acts as leader
log	Logical coordinates of this node
mincd	Minimum coordinates known by this node
maxcd	Maximum coordinates known by this node
f	Current figure displayed by the node
r	Node's calculated rotation
rc	Reset count variable

How do we model transitions between phases? As before, we introduce a monitoring process that keeps an overview of the overall status of the system. By means of communication actions, we can then allow nodes to process to the next phase: we define `phtry` | `phack` → `phcomm` with parameter

n as follows: whenever a node tries to leave phase n, it issues a `phtry(n)` action. The monitor process issues `phack(n)` actions for every phase n which may be left (i.e. $phack(1)$ means all nodes may enter phase 2). This means any node can attempt to change state at any point in time, which is exactly what we wish to analyze in our model - the monitor will introduce *soft synchronization*: phase transitions are only allowing if a condition allowed them is satisfied, but nodes have the possibility to delay transition to the next phase.

When should the monitor process issue *phack* actions? Previously, our monitoring process would issue *success* actions once the desired condition was received; we need to update this monitor so that once the end of state n is reached, the monitoring process starts issuing $phack(n)$ messages to allow any process that attempts to leave state n to succeed. This can be achieved by counting the number of nodes that need to finish phase n - if this amount is zero, $phack(n)$ can be issued.

As the original specification in [Hen08] is not clear regarding receiving messages which belong to another state, we introduce three possible scenarios and investigate the overall system behavior:

- Out of phase messages are completely ignored
  Whenever a message arrives that does not belong in the our current phase, it is discarded.
- Out of phase messages will be processed
  Any arriving message will be processed, but messages will only be sent if a node is in the correct phase to do so.
- Phasing will be completely ignored, any message can be sent/processed at any time.
  The notion of phases is removed completely, any message may be sent/processed at any time.

These scenarios shall be discussed in the next sections. The remainder of this chapter will focus on the effect of unreliable communication, adding and removing nodes in the network and illustrate a problem identified in the original specification.

Throughout this chapter, all situations presented use a 2×2 grid configuration as depicted in 10.1. It turns out a 2×2 grid is sufficient to discuss the possible issues.

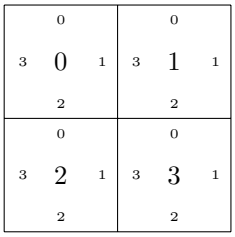


Figure 10.1: Network configuration analyzed throughout this chapter

10.1 Ignoring out of phase messages

The specifications from Chapters 6 - 9 have been merged, while strengthening all conditions with $ph =$ corresponding phase number, adding the reset phase and a different monitor. The complete source can be found in Appendix C, along with the exact renaming used. This specification results in the statespace depicted in Figure 10.2.

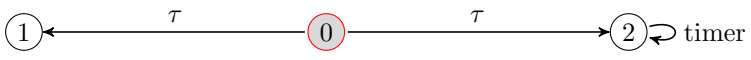


Figure 10.2: Overall system, out of phase messages are ignored

As can be seen in Figure 10.2, there are two possible paths: one path results in a deadlock, whereas the other path results in the desired functionality. This means we need to investigate why this deadlock occurs and determine whether it is possible in practice. Figure 10.3 illustrates a possible deadlock scenario in the 2×2 network configuration depicted in Figure 10.1.

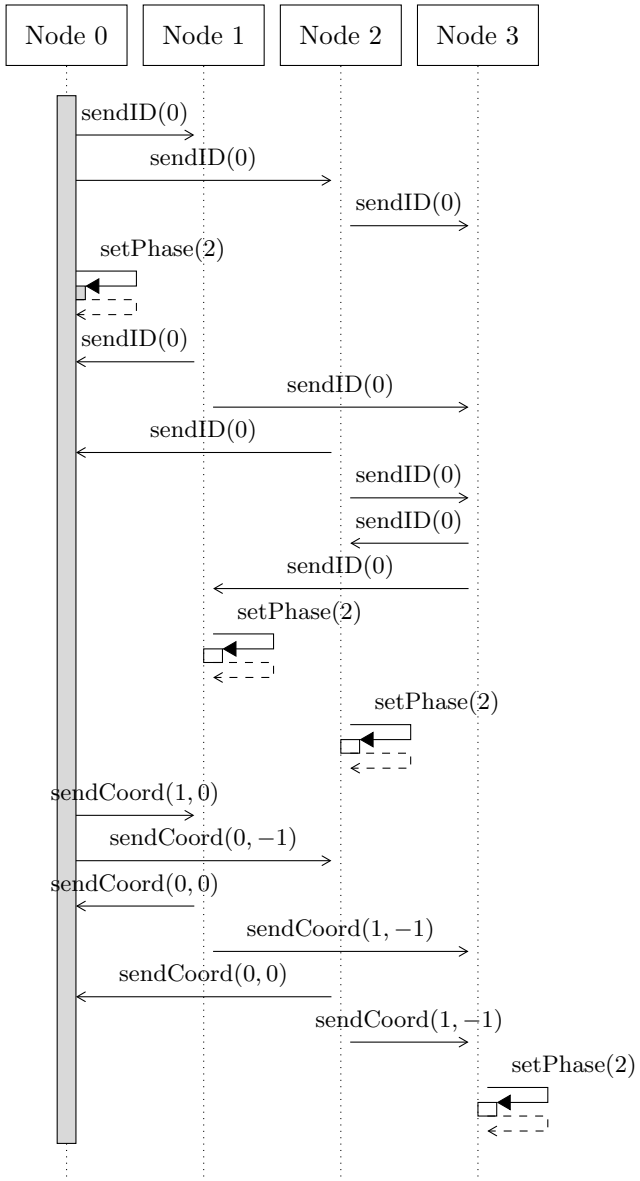


Figure 10.3: Message sequence chart of model deadlock, where time of all nodes is synchronized

There are multiple possible deadlock scenarios, but these seem to be equivalent to the one we are discussing. Refer to Figure 10.3, which is a possible scenario. We note that node 0 can enter phase 2 of the system quite rapidly: when all but one node have decided they will not become leader, the $phtry(1)$ action will become unblocked. As we can see in the model, if a node has sent to all directions, it will try to enter the next phase.

Evidently, the problem is that if a node can enter the next phase, this does not mean it will *immediately* do so - the action may be delayed. As we clearly see in the message sequence chart, should any of the nodes delay the $setPhase(2)$, it will ignore any subsequent coordinate messages. So, while the remaining nodes are waiting to be able to enter phase 3, node 3 is waiting for coordinate messages which will no longer arrive. The result is that the entire system is deadlocked.

The important question is: will this happen in practice? The answer is no, for two reasons:

- There is no monitor process in reality
  This means the system will always continue to the next phase after some time.
- We may rely on the fairness assumption
  It makes no sense to assume that a node will wait for a very long time.

This actually means our model is inadequate, as it does not reflect the real world situation. However, the main problem is that we cannot do better without introducing timing: we cannot specify that an action can only be delayed until some point in time and must be performed afterwards. The result is that we cannot conclude that the algorithms will always work as expected; we can only do so under the *fairness assumption*. Informally speaking, the fairness assumption means eventually something good will happen; if an action is possible, we expect that it will eventually be performed; it will not happen that a node suddenly waits for a long time before continuing. This is precisely the condition we need to prevent the scenario in Figure 10.3: if we assume no node will delay for a long period of time, the scenario will not occur and the system will indeed perform as expected.

10.2 Processing out of phase messages

In order to process out of phase messages, we need to alter the model used in the previous section by always accepting a message no matter what; this means we need to remove the $ph =$ number conditions before all $recv\ldots$ actions. The resulting statespace can be found in Figure 10.4.

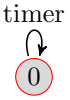


Figure 10.4: Overall system, out of phase messages are accepted

As is clearly visible in Figure 10.4, there are no deadlocks present: the system keeps performing `timer` actions. Since there are no `reset` actions, we can conclude that accepting all messages benefits the system: it will not result in sudden resets, and the response time can only get better. Of course, it must be noted that the monitor process keeps some sense of regularity between processes, but the state space clearly illustrates that accepting any message will not result in sudden reset cycles.

10.3 Ignoring the complete notion of phasing

Before we alter the model, we note that we cannot completely remove the current phasing parameter from the model. The reason is that even though we disregard phasing, we want to ensure that every event noticed by the monitoring process (for example, all nodes but one decide they will not become leader, which is used to allow transition to the next phase) must occur only *once* per node, as our proposed algorithms run only once. The result is that within the model, all $ph =$ number lines are removed unless these are present for `phtry(n)` actions. The result is that messages are sent/received in any phase of the system, yet each phase is only initiated once.

Unfortunately, this results in an extremely large system - even while trying to analyze a 2×2 network, there are millions of states and hundreds of millions of transitions, which cannot immediately be reduced. In fact, the resulting system is so large that it cannot be analyzed anymore using the toolset; this is simply is not feasible.

However, we can apply some reasoning to the problem and state our expectations. We expect that completely ignoring phasing is a very bad idea: any node could immediately attempt to send status update messages containing its own identification number. This can often trigger a reset, causing the entire network to reset again - where the exact same situation can arise again and again.

10.4 Effect of unreliable communication

In this section, we will alter the model in such a way that a single node in the network (which we will refer to as the faulty node $\mathcal{F}$) is unreliable: messages sent to this node do not have to arrive. Recall that we have `send...` and `intern` actions, where the latter represents the act of sending a message that will not be received. As we intend to provide the *possibility* that a message gets lost, it makes sense to alter sending a message of each phase to (where `FAULTYNUM` $= \mathcal{F}$):

```
190 (s<4&&(neighbornum(num,rr(s,RotationList.num))==num||
191        neighbornum(num,rr(s,RotationList.num))==FAULTYNUM)) ->
192     intern . Smartpixel(...,s+1,...) +
193 (s<4&&(neighbornum(num,rr(s,RotationList.num))!=num)) ->
194     send...(neighbornum(num,rr(s,RotationList.num)),...) . Smartpixel(...,s+1,...)
```

Note that we also have to alter the monitor process: the monitor knows that node $\mathcal{F}$ is unreliable, so it cannot expect that this node will ever achieve the desired status, thus any updates send to the monitor by node $\mathcal{F}$ must not influence the monitor status.

The resulting statespace is quite large, and there do not seem to be any obvious patterns there. However, the main point of interest is: is there a deadlock? This can be expressed by the modal formula $[true^\star]\langle true\rangle true$. Unfortunately, the modal formula analysis tools of mCRL2 are currently not powerful enough to prove or disprove this statement. Even when experimental tools which attempt to reduce the number of calculations needed are used, this still appears to be infeasible. However, it is possible to generate a state space, which is depicted in Figure 10.5.

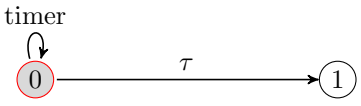


Figure 10.5: Overall system, node 1 may miss any message

The *timer* loop makes sense: if the leader node considers itself to be ready, it may continue to instruct any surrounding nodes to change their figure, which they will do. However, where does the deadlock come from? It turns out this is the result of multiple leaders becoming active, as is depicted in Figure 10.6.

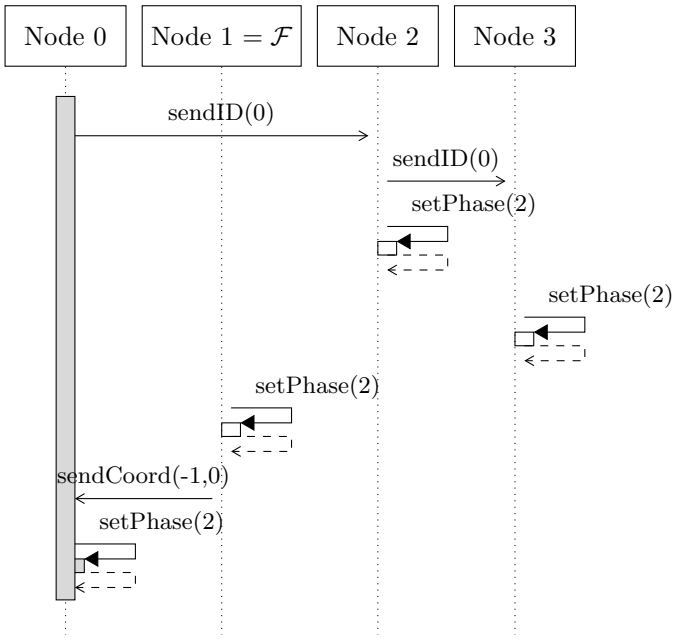


Figure 10.6: Message sequence chart detailing monitor issues if node $\mathcal{F}$ doesn't have reliable communication, where time of all nodes is synchronized

As illustrated in 10.6, a possible scenario is that node $\mathcal{F}$ never receives word of a lower leader id, causing it to assume that it should become leader itself. However, node 0 will eventually also consider itself to be leader, as it truly has the lowest id. The deadlock shows up when node 0 delays becoming leader long enough for node $\mathcal{F}$ to start issuing coordinate messages. As node 0 doesn't consider itself to be leader, it will obey these messages. Yet, when it decides it will become leader, it considers itself to be at $\langle -1, 0 \rangle$ as it accepted these coordinates while it was still waiting to become leader. The monitoring process will not acknowledge these coordinates and other nodes will not notice any conflicting coordinates, resulting in the deadlock.

Will this issue occur in reality? Based on the fairness assumption, we are inclined to claim it will not - yet, if a node is switched on/off at an ill time, this condition may very well occur. Will the actual system deadlock in such a case? The answer is no: the actual system will keep on continuing since nothing seems to be wrong. Only once the two leaders are issuing updates, the remaining nodes will detect that there are multiple leaders and in turn reset the system.

However, we can prevent this issue by requiring a stronger condition to hold when a node intends to become leader. If a node has already received coordinates before it decided to become leader, it knows another node is already acting as leader; it would not have gotten these coordinates otherwise. The result is that a reset is to be issued. If a node did not receive any coordinates, it should just continue assuming leadership as usual. Thus, it makes sense to update the specification to:

```
195 (ph==1&&sl==4&&mapNodeId(num)==leader_id&&log==coord(0,0)) ->
196     phtry(1) . becomeLeader(num) .
197      Smartpixel(num,leader_id,2,sl,0,ss,su,sr,true,faulty,log,mincd,maxcd,f,r,rc) +
198 (ph==1&&sl==4&&mapNodeId(num)==leader_id&&log!=coord(0,0)) ->
199     phtry(1) . sendMonReset .
200      Smartpixelreset(num,rc) +
```

As this will cause the system to issue a reset once the upcoming leader realizes it has already received coordinates, which it can only have received by a node which has already assumed leader-

ship. We have analyzed a model with this modification, which will become rather large as system resets can be triggered at quite a few occasions. In fact, attempting to show the resets will result in an enormous statespace, which is incomprehensible. However, if we only show the `timer` action, we obtain the state space as illustrated in Figure 10.7.

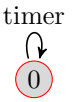


Figure 10.7: Overall updated system, node $\mathcal{F}$ may miss any message

It is clear by inspection of Figure 10.7 that the deadlock no longer occurs, which shows faulty nodes will not cause the system to deadlock.

10.5 Effect of adding/removing nodes

Since it is not possible to dynamically add/remove processes in the mCRL2 language (processes can only be instantiated a single time), we added a parameter to indicate whether a node is physically present. Initially, all nodes but a single node are present, while a node we shall refer to as $\mathcal{P}$ could become present (this event will be called the *activation* of $\mathcal{P}$) at any time.

However, this approach yields severe problems when trying to alter the monitoring process to deal with the new situation. The main issue is that since $\mathcal{P}$ can become active at any time, the monitor must immediately be aware of this new situation. Yet, this may cause a conflict with the previous situation. Let us illustrate this by a Message Sequence Chart as depicted in Figure 10.8.

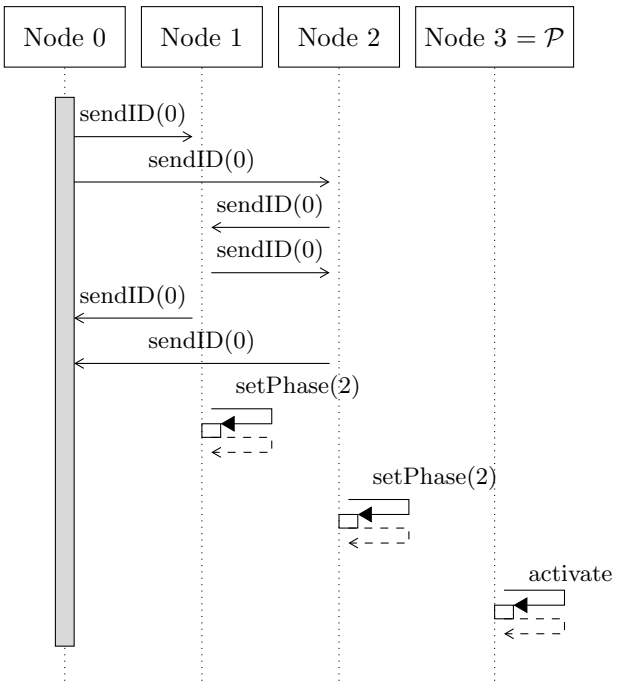


Figure 10.8: Message sequence chart detailing monitor issues if a new node $\mathcal{P}$ can be activated at any time, where time of all nodes is synchronized

It is possible that node $\mathcal{P}$ is activated while some nodes have processed to phase 2, while other nodes have not. The result is that both node 0 and node $\mathcal{P}$ will be blocked from entering phase

2, as the conditions to allow them to proceed to the next phase are no longer satisfied. Since we cannot adequately model time, as we are in an untimed model, it is not possible to analyze what the result on the system is - in reality, the system would just continue and after some time notice that both node 0 and node $\mathcal{P}$ act as leader, resulting in a reset and thus a new leader election.

10.6 Issues identified while analyzing the overall system

Throughout this chapter, it appears no issues have been identified once all phases of the system have been merged in one single model. However, in the original specification as described in [Hen08], if any node receives some leader id which is not equal to the leader id the node has stored, a `reset` would be initiated. The resulting statespace of such a model is illustrated in Figure 10.9. Clearly, a reset is very undesirable: this indicates that in ideal circumstances (i.e. the network configuration is not being altered and node communication is 100% reliable) the system can decide to start from scratch again, which should only be performed once a node believes there are multiple leaders. As this clearly should not be possible in our system, we need to investigate exactly why this reset is triggered.

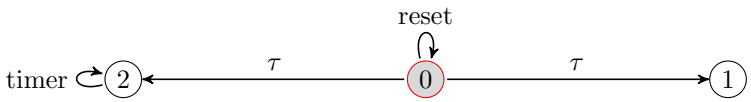


Figure 10.9: Overall system, out of phase messages are ignored and any non-matching leader id results in a reset

After inspection, it turns out the problem is due to a scenario as depicted in Figure 10.10. During the first phase, a leader is elected, and phase two is unblocked when $N - 1$ nodes decide they will not become leader. However, this does not mean all nodes agree upon the same leader id. This may seem in contrast to Property 6.2; however, this property only claims that if we wait long enough, all nodes will *eventually* all obtain the same leader id. Since node speeds are not synchronized, this may not happen as some nodes may leave the leadership election phase sooner than others, which is exactly what we see in Figure 10.10; it's quite possible a node is lagging behind or has joined in rather late, and thus only knows some other node will become leader but it may have the wrong node id as the leader. This shouldn't be a problem, since the leader election should result in a single node assuming leader status.

However, during the minimum/maximum coordinate determination phase of the algorithm, the leader id is sent along to determine whether two different leaders are determining min/max coordinates. Thus, if different leader id's are rejected, we implicitly assume the entire network was already aware of the actual leader id - and as can be seen in the message sequence chart, this doesn't have to be the case.

In order to prevent this unwanted condition, we introduced two alterations of the protocol:

- During minimum/maximum coordinate determination, if a lower leader id is received than what is currently known, that id will be used instead.
  Any higher id will still cause a reset, though.
- During the showtime phase, any node that considers itself to be leader will always send its own id number.
  This ensures there is only one leader, since there can be only one smallest id number.

As can be seen in Section 10.1, this modification of the protocol prevents the unwanted `reset` actions. And, more importantly, our model has illustrated a scenario in which our seemingly correct protocols would not result in the intended behavior, illustrating the benefit of formal modeling.

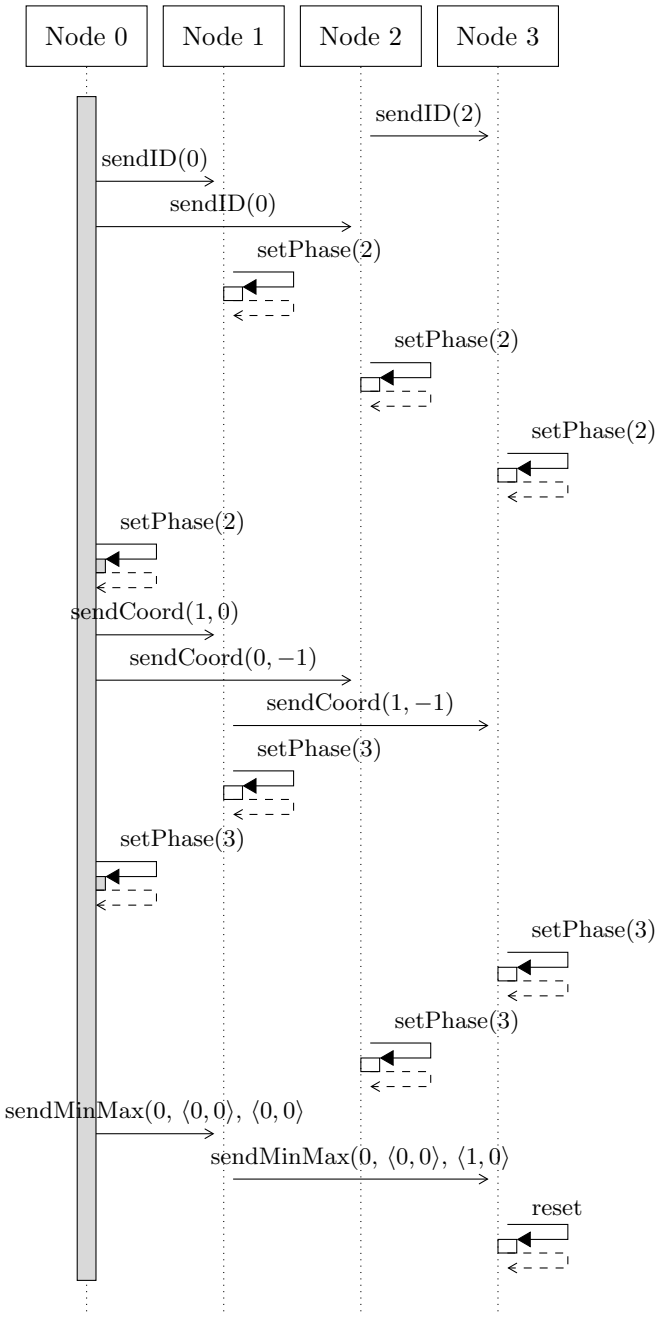


Figure 10.10: Message sequence chart illustrated `reset` behavior of Figure 10.9

Chapter 11

Timed to untimed conversion

In the previous chapters, the protocol has been verified without explicitly modeling timing requirements. However, it is possible to transform timed μCRL to untimed μCRL without affecting the behavior of the system; a technique to do so is described in [RGvdZvW02], where it is used to construct a completeness proof. In this chapter, we discuss an approach to transform timed mCRL2 processes to untimed mCRL2 processes illustrating similar behavior. This allows us to investigate the influence of time on the system, while still being able to use the mCRL2 toolset.

11.1 Theory of timed and untimed processes

First of all, we introduce $\mathbb{T}$, which is our time domain. The only requirements on $\mathbb{T}$ are that it has a total ordering, referred to as $\leq$, and a least element which we will refer to as 0. We will introduce a timed linear process equation, or TLPE, which is the format we will use as basic 'building blocks', as it is possible to transform any given mCRL2 specification to this basic format using a process known as *lineralization*. However, this transformation is a separate research subject, which is discussed in detail in [Use02].

Definition 11.1 (Timed Linear Process Equation)**.** A timed linear process equation (TLPE) is a process of the following form:

$$X(d:D) = \sum_{i \in I, e_i : E_i} c_i(d, e_i) \rightarrow \alpha_i(d, e_i) \cdot t_i(d, e_i) \cdot X(g_i(d, e_i))$$

Where D is some set representing the data parameters, I is a finite index set and for all $i \in I$:

- $c_i(d, e_i)$ is a condition
- $\alpha_i(d, e_i)$ is a multi-action $a_i^1(f_i^1(d, e_i)) \mid \ldots \mid a_i^{n_i}(f_i^{n_i}(d, e_i))$, where $f_j^k(d, e_j)$ give the parameters of action a_j^k, for all $1 \leq k \leq n_i$.
- $t_i : D \times E_i \rightarrow \mathbb{T}$ are the timestamps of multi-action $\alpha_i(d, e_i)$
- $g_i : D \times E_i \rightarrow D$ defines process parameters in the next state.

An important aspect of a TLPE is that the parallel composition of two TLPE's can always be reduced to the shape of a TLPE, as shown in Theorem 11.2.

Theorem 11.2. *For TLPE's X, Y and Z where $Z(d_i, d_j) = X(d_i) \parallel Y(d_j)$, it holds that:*

$$\begin{aligned} Z(d_i,d_j) \quad = \quad & \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) < t_j(d_j,e_j) \rightarrow \\ & \alpha_i(d_i,e_i) \lhd t_i(d_i,e_i) \cdot Z(g_i(d_i,e_i),d_j) + \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_j(d_j,e_j) < t_i(d_i,e_i) \rightarrow \\ & \alpha_j(d_j,e_j) \lhd t_j(d_j,e_j) \cdot Z(d_i,g_j(d_j,e_j)) + \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) = t_j(d_j,e_j) \rightarrow \\ & \alpha_i(d_i,e_i) \mid \alpha_j(d_j,e_j) \lhd t_i(d_i,e_i) \cdot Z(g_i(d_i,e_i),g_j(d_j,e_j)) + \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \rightarrow \\ & \delta \lhd min(t_i(d_i,e_i),t_j(d_j,e_j)) \cdot Z(d_i,d_j) \end{aligned}$$

Proof. Refer to Appendix A □

Additionally, we define a deadlock saturated linear process as follows:

Definition 11.3 (Deadlock Saturated Timed Linear Process Equation)**.** A deadlock saturated timed linear process equation (DS-TLPE) is a process of the following form:

$$\begin{aligned} X(d:D) \quad = \quad & \textstyle\sum_{i\in I,e_i:E_i} c_i(d,e_i) \rightarrow \alpha_i(d,e_i) \lhd t_i(d,e_i) \cdot X(g_i(d,e_i)) + \\ & \textstyle\sum_{u:\mathbb{T}} c_i(d,e_i) \wedge u < t_i(d,e_i) \rightarrow \delta \lhd u \cdot X(g_i(d,e_i)) \end{aligned}$$

Where D, I, α_i, c_i, t_i and g_i are defined similar to Definition 11.1

We note that it is possible to transform any TLPE to a DS-TLPE and the other way around, as a result of Theorem 11.4.

Theorem 11.4. *For any TLPE X, there exists a DS-TLPE Y such that $X = Y$.*

Proof. Refer to Appendix A □

The reason we distinguish between a TLPE and a DS-TLPE is because we can transform the latter directly to an untimed specification while preserving the process behavior, provided that the process is *welltimed*. The intuition of welltimedness arises by the following observation: in a timed environment the identity $a \lhd 3 \cdot (b \lhd 2 + c \lhd 4) = a \lhd 3 \cdot c \lhd 4$ holds: if an action a has been performed at time interval 3, it impossible to subsequently perform a b action at time interval 2 without grossly violating the laws of physics. However, if we were to transform this expression to a TLPE, we obtain $a(3) \cdot (b(2) + c(4))$, which is not equal to $a(3) \cdot c(4)$. In order to prevent this unwanted behavior, we restrict the TLPE to a form where the first expression cannot occur. More formally, this is defined as follows:

Definition 11.5. A TLPE $X(d : D)$ is welltimed if and only if

$$\forall_{i\in I,j\in I,e_i:E_i,e_j:E_j} c_i(d,e_i) \wedge c_j(g_i(d,e_i),e_j) \rightarrow t_i(d,e_i) < t_j(g_i(d,e_i),e_j)$$

Finally, we define a time-free linear process equation (TF-LPE) as follows:

Definition 11.6 (Time-free Linear Process Equation)**.** A time-free linear process equation (TF-LPE) is a process of the following form:

$$X(d:D) = \sum_{i\in I,e_i:E_i} c_i(d,e_i) \rightarrow \alpha_i(d,e_i,t_i(d,e_i)) \cdot X(g_i(d,e_i)) \; + \sum_{j\in J,e_j:E_j} c_j(d,e_j) \rightarrow \Delta(t_j(d,e_j))$$

Where D, I, α_i, c_i, t_i and g_i are defined similar to Definition 11.1. J is a finite index set and for all $j \in J$:

- $c_j(d, e_j)$ is a condition

- $\Delta : \mathbb{T}$ is an action representing the deadlock $\delta \triangleleft t$.
- $t_j : D \times E_i \to \mathbb{T}$ are the timestamps of action Δ

By using the results of [RGvdZvW02], it is possible to transform a welltimed DS-TLPE to a TF-LPE. To fulfill this goal, a tool called LeMans has been developed. This tool will be described in the next section.

11.2 Introducing the LeMans tool

Using the theory outlined in the previous chapter, we have developed a tool called LeMans. The purpose of this tool is to transform a welltimed TLPE to a TF-LPE, which can then be used in the mCRL2 toolset. An input must be in the following shape:

$$\begin{array}{ll} \text{proc} & P(\dots) = \dots \\ \dots & \\ \text{proc} & Z(\dots) = \dots \\ & \\ \text{init} & P(\dots) \parallel \dots \parallel Z(\dots) \end{array}$$

Which results in the following output:

$$\begin{array}{ll} \text{proc} & P(\overrightarrow{a}) = \dots \\ \text{init} & P(\mathcal{I}_P) \end{array}$$

Where P is a TF-LPE. This is implemented using the following sequence of actions:

1. One or more TLPE's are read from the standard input
   We will refer to the right hand side of process P as $\mathcal{T}_P$.
2. A process initializer of the form **init** $P(\dots) \parallel \dots \parallel Z(\dots)$ is read from the standard input. We will refer to the process initializer as $\mathcal{I}$.
3. Arguments of $\mathcal{I}$ are determined.
   Since $\mathcal{I}$ consists of multiple processes in parallel, the initialization parameters of all these processes must be combined to create a suitable initializer for the to-be-constructed TLPE. These parameters will be referred to as $\overrightarrow{a}$.
4. The right hand sides of the TLPE processes are filled out in $\mathcal{I}$
   $\mathcal{I}$ becomes $\mathcal{T}_P(\dots) \parallel \dots \parallel \mathcal{T}_Z(\dots)$.
5. By repeatably applying Theorem 11.2, a single TLPE P is obtained, which is derivably equal to $\mathcal{I}$.
6. The TLPE P is transformed to a DS-TLPE by introducing extra summands
   This can be done automatically as a result of Theorem 11.4.
7. Time-free abstraction is applied to P
   This can be done automatically by applying [RGvdZvW02, Definition 4.4].
8. SUM1-elimination is applied to P
   Any expression of the form $\sum_v X$ where v does not appear in X will be rewritten to X. This is necessary to avoid unbounded expansion in the `lps2lts` tool

The result of the LeMans tool is a mCRL2 specification suitable for use in the toolset.

11.3 Results on leader election

The LeMans tool has been applied on the following specification, which is the leader election algorithm as presented in Section 4.1. We define $\mathcal{E} = 2$ as the amount of time it may take to transmit a single message, and $\Omega = 10$ as the amount of time the leader election algorithm waits after the last incoming message before terminating.

proc $P(id, lid, s : \mathbb{N}, t, u : \mathbb{T}, done : \mathbb{B}) =$
$\sum_{v:\mathbb{T}}\ 0 < v \leq \mathcal{E} \wedge s < 4 \wedge neighbor(id, s) \neq id \rightarrow sendID(neighbor(id, s), lid)^{\lhd}(t + v) \cdot$
$P(id, lid, s + 1, t + v, u, done) +$
$\sum_{v:\mathbb{T}}\ 0 < v \leq \mathcal{E} \wedge s < 4 \wedge neighbor(id, s) = id \rightarrow intern^{\lhd}(t + v) \cdot$
$P(id, lid, s + 1, t + v, u, done) +$
$\sum_{v:\mathbb{T}, lid':\mathbb{N}}\ 0 < v \leq \mathcal{E} \wedge lid' < lid \rightarrow recvID(id, lid')^{\lhd}(t + v) \cdot$
$P(id, lid', 0, t + v, t + v + \Omega, done) +$
$\sum_{v:\mathbb{T}, lid':\mathbb{N}}\ 0 < v \leq \mathcal{E} \wedge lid' \geq lid \rightarrow recvID(id, lid')^{\lhd}(t + v) \cdot P(id, lid, s, t + v, u, done) +$
$\sum_{v:\mathbb{T}}\ s = 4 \wedge id = lid \wedge 0 < v \leq \mathcal{E} \wedge \neg done \rightarrow leader(id)^{\lhd}(u + v) \cdot P(id, lid, s, u + v, u, true) +$
$\sum_{v:\mathbb{T}}\ s = 4 \wedge id \neq lid \wedge 0 < v \leq \mathcal{E} \wedge \neg done \rightarrow normal(id)^{\lhd}(u + v) \cdot P(id, lid, s, u + v, u, true)$

init $P(0, 0, 0, 0, \Omega, false) \parallel P(1, 1, 0, 0, \Omega, false) \parallel$
$P(2, 2, 0, 0, \Omega, false) \parallel P(3, 3, 0, 0, \Omega, false)$

Process P is obviously a TLPE. It is not immediately obvious that process P is welltimed. We support the claim of welltimedness by claiming that for every action that is performed at some time interval v, the t parameter of the process is updated to v. The end result is that time will always increase in the process, which has the result that process P is welltimed as claimed.

Using the LeMans tool on this input results in TF-LPE output, which can be used as mCRL2 toolset input. However, trying to process the TF-LPE (which consists of roughly 140.000 summands) was not possible at the time of writing: the `mcrl22lps` tool does not appear to terminate while linearalising the TF-LPE input[1]. This has prevented us from doing any in-depth analysis on the now untimed mCRL2 models.

[1]This issue has been reported to the mCRL2 developers, but as of writing, no solution was available

Chapter 12

Closing words

Throughout the thesis, we have described a complete specification of the SmartPixel system and presented a model of this specification. We have used the mCRL2 toolset in order to validate these models, initially on a per-algorithm basis while later combining all algorithms into a single system. During the analysis of this combined system, it turned out that attempting to analyze such an inherently timed system with a toolset that currently does not support timing is challenging and at times pretty cumbersome.

However, we can conclude that the algorithms as presented in Chapter 4 are proved to be correct (and can be improved by applying the modifications as outlined in Section 10.4). Great effort has been taken to show how models illustrating the behavior of these protocols were constructed piece by piece, with the intention of showing how such a model can be created from solely an informal specification. Later on, these models were combined and the interactions between the various algorithms were studied, something the informal specification was especially vague on.

The result is that all ambiguities of the original specification have been resolved and the resulting specification indeed performs as intended as shown in Chapter 10 in a fixed 3×3 network, which we believe to be adequate to show system correctness. Furthermore, the correctness of all individual algorithms have been proved in both 3×3 and 4×4 network configurations. During the verification, at least one major design problem in the original specification has been found and resolved, which would have otherwise gone unnoticed. We have attempted a timed analysis, with some success using the UPPAAL toolset even though it was not usable for large specifications. An attempt to convert a timed model to an untimed model was successful, but this did not provide any fruitful results; the mCRL2 toolset was never designed for automatically generated specifications. This becomes especially clear as it seems unable to process the resulting untimed specification.

During the validation process, a lot of problems in the mCRL2 toolset have been discovered. All serious showstopper issues, like the inability to compare negative numbers, have been resolved quite adequately. However, any request to make the toolset more pleasant to use has been dismissed. While the developers are to be commended for the sheer dedication and response times on bugs, it is our belief that especially for bigger specifications of real-life systems such as the SmartPixel system, the toolset would really benefit from more syntactic sugar: for example, while working on the specification that is provided as Appendix C, a lot of time was spent analyzing what ultimately was a typo in the parameter list - only a single parameter needed to be changed, yet the syntax requires the entire process parameters to be duplicated, which is quite tedious and error-prone. Another request that was turned down is the ability to generate the process initializer; i.e. if the network is 3×3, an initializer consisting of $SmartPixel(0) \parallel \cdots \parallel SmartPixel(8)$ would automatically be created. Small features such as these would really be beneficial to users of the toolset.

Another aspect that will greatly benefit the toolset is improved documentation. A subset of the reported issues were the result of problems that are known to the specific tool developer, but not to all other developers. While some of these have been added to the 'Known Limitations' page on the mCRL2 website, it would be beneficial to synchronize these issues among the mCRL2 developers. An example of such an issue is the inability to generate a statespace file larger than 4GB using the default output format, which has since been added to the website.

It must be noted that the mCRL2 toolset is quite powerful. Even though it has only been compared with UPPAAL during this project, it quickly became evident that mCRL2 is capable of handling much larger models than UPPAAL: an attempt to validate leader election in a 4×4 grid is feasible in mCRL2, while 3×3 appeared barely manageable in UPPAAL. Of course, mCRL2 does not consider the notion of time like UPPAAL does, but it is clear to us that mCRL2 handles big specifications much better than UPPAAL currently does; we cannot say whether this is the result of the lack of time in mCRL2.

Finally, we would like to suggest that the mCRL2 toolset be given more functionality regarding timed analysis. Ideally, this would mean standard support for any timed specification, but this is most likely infeasible on the short term. A more realistic approach could be to integrate the LeMans tool as discussed in Section 11.2 in the toolset to transform a timed specification into an untimed specification that can be used for analysis, which would still have the advantage of being able to completely reuse mCRL2's powerful model analysis abilities. The ability to transform timed processes to untimed processes is especially useful since analysis by means of monitoring processes by no means solves all possible issues, plus that the construction of such a monitor is a challenge on its own.

Appendix A

Proofs

This appendix contains all derivations and proofs used throughout the thesis.

Lemma A.1. $(a \cdot b) \mid c = a \cdot (b \cdot c + c \cdot b + b \mid c) + c \cdot a \cdot b + a \mid c \cdot b$

Proof.

$$
\begin{aligned}
& (a \cdot b) \mid c \\
= & \{\ M\ \} \\
& (a \cdot b) \between c + c \between (a \cdot b) + a \cdot b \mid c \\
= & \{\ LM3, LM1, S5\ \} \\
& (a \ll c) \cdot (b \mid c) + (c \ll (a \cdot b)) \cdot (a \cdot b) + a \mid c \cdot b \\
= & \{\ TB1, M\ \} \\
& a \cdot (b \between c + c \between b + b \mid c) + ((c \ll a) \cdot (a \cdot b)) + a \mid c \cdot b \\
= & \{\ LM1, TB1\ \} \\
& a \cdot ((b \ll c) \cdot c + (c \ll b) \cdot b + b \mid c) + c \cdot a \cdot b + a \mid c \cdot b \\
= & \{\ TB1\ \} \\
& a \cdot (b \cdot c + c \cdot b + b \mid c) + c \cdot a \cdot b + a \mid c \cdot b
\end{aligned}
$$

□

Lemma A.2. $\delta \lhd 0 \between X = \delta \lhd 0$

Proof.

$$
\begin{aligned}
& \delta \lhd 0 \between X \\
= & \{\ \text{LM6}\ \} \\
& (\delta \between X) \lhd 0 \\
= & \{\ \text{T5}\ \} \\
& 0 > 0 \rightarrow (\delta \between X) \lhd 0 \\
= & \{\ \text{T2}\ \} \\
& 0 > 0 \rightarrow (\delta \between X) \lhd 0 \diamond \delta \lhd 0 \\
= & \{\ \text{Cond2}\ \} \\
& \delta \lhd 0
\end{aligned}
$$

□

Lemma A.3. $\delta \lhd 0 \mid X = \delta \lhd 0$

Proof. Similar to proof of Lemma A.2, using axiom S9. □

Lemma A.4. $\alpha \ll \delta \triangleleft 0 = \delta \triangleleft 0$

Proof.

$$
\begin{aligned}
& \alpha \ll \delta \triangleleft 0 \\
= & \{ \text{ TB3 } \} \\
& \textstyle\sum_{u:\mathbb{T}} u < 0 \rightarrow (\alpha \triangleleft u) \ll \delta \\
= & \{ \text{ T2 } \} \\
& \textstyle\sum_{u:\mathbb{T}} u < 0 \rightarrow (\alpha \triangleleft u) \ll \delta \diamond \delta \triangleleft 0 \\
= & \{ \text{ Cond2 } \} \\
& \delta \triangleleft 0
\end{aligned}
$$

□

Lemma A.5. $\delta \triangleleft u \cdot X = \delta \triangleleft u$

Proof.

$$
\begin{aligned}
& \delta \triangleleft u \cdot X \\
= & \{ \text{ TA4 } \} \\
& (\delta \cdot X) \triangleleft u \\
= & \{ \text{ A7 } \} \\
& \delta \triangleleft u
\end{aligned}
$$

□

Lemma A.6. $X \parallel Y = Y \parallel X$

Proof.

$$
\begin{aligned}
& X \parallel Y \\
= & \{ \text{ M } \} \\
& X \between Y + Y \between X + X \mid Y \\
= & \{ \text{ S1 } \} \\
& X \between Y + Y \between X + Y \mid X \\
= & \{ \text{ A1 } \} \\
& Y \between X + X \between Y + Y \mid X \\
= & \{ \text{ M } \} \\
& Y \parallel X
\end{aligned}
$$

□

Lemma A.7. *For TLPE's X and Y, it holds that*

$$
\begin{aligned}
X(d_i) \between Y(d_j) \quad = \quad & \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) < t_j(d_j, e_j) \rightarrow \\
& \qquad \alpha_i(d_i, e_i) \triangleleft t_i(d_i, e_i) \cdot (X(g_i(d_i, e_i)) \parallel Y(d_j)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) \geq t_j(d_j, e_j) \rightarrow \\
& \qquad \delta \triangleleft min(t_i(d_i, e_i), t_j(d_j, e_j)) \cdot (X(d_i) \parallel Y(d_j))
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& X(d_i) \between Y(d_j) \\
= \{\ & \text{definition } X \ \} \\
& (\textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \cdot X(g_i(d_i,e_i))) \between Y(d_j) \\
= \{\ & \text{LM5} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \cdot X(g_i(d_i,e_i)) \between Y(d_j) \\
= \{\ & \text{T2} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \cdot X(g_i(d_i,e_i)) \diamond \delta \triangleleft 0 \between Y(d_j) \\
= \{\ & \text{Cond} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \cdot X(g_i(d_i,e_i)) \between Y(d_j) + \neg c_i(d_i,e_i) \to \delta \triangleleft 0 \between Y(d_j) \\
= \{\ & \text{Lemma A.2, T1} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \cdot X(g_i(d_i,e_i)) \between Y(d_j) \\
= \{\ & \text{LM3} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to (\alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \ll Y(d_j)) \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \\
= \{\ & \text{definition } Y \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to (\alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \ll (\sum_{j\in I_y,e_j:E_j} c_j(d_j,e_j) \to \alpha_j(d_j,e_j) \triangleleft t_j(d_j,e_j) \cdot Y(g_j(d_j,e_j)))) \\
& \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \\
= \{\ & \text{TB6} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to (\sum_{j\in I_y,e_j:E_j} \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \ll (c_j(d_j,e_j) \to \alpha_j(d_j,e_j) \triangleleft t_j(d_j,e_j) \cdot Y(g_j(d_j,e_j)))) \\
& \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \\
= \{\ & \text{T2, TB5} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to (\sum_{j\in I_y,e_j:E_j} \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \ll (c_j(d_j,e_j) \to \alpha_j(d_j,e_j) \triangleleft t_j(d_j,e_j) \diamond \delta \triangleleft 0)) \\
& \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \\
= \{\ & \text{Cond} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to (\sum_{j\in I_y,e_j:E_j} c_j(d_j,e_j) \to \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \ll \alpha_j(d_j,e_j) \triangleleft t_j(d_j,e_j) \\
& + \neg c_j(d_j,e_j) \to \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \ll \delta \triangleleft 0) \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \\
= \{\ & \text{Lemma A.4, T1} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \to (\sum_{j\in I_y,e_j:E_j} c_j(d_j,e_j) \to \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \ll \alpha_j(d_j,e_j) \triangleleft t_j(d_j,e_j)) \\
& \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \\
= \{\ & \text{SUM4, Cond} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \to (\alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \ll \alpha_j(d_j,e_j) \triangleleft t_j(d_j,e_j)) \\
& \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \\
= \{\ & \text{TB3, TB1} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \to (\sum_{u:\mathbb{T}} u < t_j(d_j,e_j) \to \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \triangleleft u) \\
& \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \\
= \{\ & \text{TA1} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \to (\sum_{u:\mathbb{T}} u < t_j(d_j,e_j) \to ((t_i(d_i,e_i) = u) \to \\
& \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \diamond \delta \triangleleft min(t_i(d_i,e_i),u))) \\
= \{\ & \exists_{u:\mathbb{T}} u < t_j(d_j,e_j) \wedge t_i(d_i,e_i) = u \equiv t_i(d_i,e_i) < t_j(d_j,e_j), \text{ SUM1} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \to (t_i(d_i,e_i) < t_j(d_j,e_j) \to \\
& \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) \diamond \delta \triangleleft min(t_i(d_i,e_i),t_j(d_j,e_j))) \\
= \{\ & \text{Cond} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) < t_j(d_j,e_j) \to \\
& \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) + \\
& \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) \geq t_j(d_j,e_j) \to \\
& \delta \triangleleft min(t_i(d_i,e_i),t_j(d_j,e_j)) \\
= \{\ & \text{Lemma A.5} \ \} \\
& \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) < t_j(d_j,e_j) \to \\
& \alpha_i(d_i,e_i) \triangleleft t_i(d_i,e_i) \cdot (X(g_i(d_i,e_i)) \parallel Y(d_j)) + \\
& \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) \geq t_j(d_j,e_j) \to \\
& \delta \triangleleft min(t_i(d_i,e_i),t_j(d_j,e_j)) \cdot (X(d_i) \parallel Y(d_j))
\end{aligned}
$$

□

Lemma A.8. *For TLPE's X and Y, it holds that:*

$$\begin{aligned} X(d_i) \mid Y(d_j) \quad = \quad & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) = t_j(d_j,e_j) \rightarrow \\ & \alpha_i(d_i,e_i) \mid \alpha_j(d_j,e_j) \lhd t_i(d_i,e_i) \cdot (X(g_i(d_i,e_i)) \parallel Y(g_j(d_j,e_j))) + \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) \neq t_j(d_j,e_j) \rightarrow \\ & \delta \lhd min(t_i(d_i,e_i),t_j(d_j,e_j)) \cdot (X(d_i) \parallel Y(d_j)) \end{aligned}$$

Proof.

$$\begin{aligned} & X(d_i) \mid Y(d_j) \\ = \; & \{ \text{ definition } X \text{ and } Y \; \} \\ & (\textstyle\sum_{i\in I_x,e_i:E_i} c_i(d_i,e_i) \rightarrow \alpha_i(d_i,e_i) \lhd t_i(d_i,e_i) \cdot X(g_i(d_i,e_i))) \mid \\ & (\textstyle\sum_{j\in I_y,e_j:E_j} c_j(d_j,e_j) \rightarrow \alpha_j(d_j,e_j) \lhd t_j(d_j,e_j) \cdot Y(g_j(d_j,e_j))) \\ = \; & \{ \text{ S8, S1, SUM4 } \} \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} (c_i(d_i,e_i) \rightarrow \alpha_i(d_i,e_i) \lhd t_i(d_i,e_i) \cdot X(g_i(d_i,e_i))) \mid \\ & (c_j(d_j,e_j) \rightarrow \alpha_j(d_j,e_j) \lhd t_j(d_j,e_j) \cdot Y(g_j(d_j,e_j))) \\ = \; & \{ \text{ T2 } \} \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} (c_i(d_i,e_i) \rightarrow \alpha_i(d_i,e_i) \lhd t_i(d_i,e_i) \cdot X(g_i(d_i,e_i)) \diamond \delta \lhd 0) \mid \\ & (c_j(d_j,e_j) \rightarrow \alpha_j(d_j,e_j) \lhd t_j(d_j,e_j) \cdot Y(g_j(d_j,e_j)) \diamond \delta \lhd 0) \\ = \; & \{ \text{ Cond } \} \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} (c_i(d_i,e_i) \rightarrow \alpha_i(d_i,e_i) \lhd t_i(d_i,e_i) \cdot X(g_i(d_i,e_i)) \diamond \delta \lhd 0) \mid \\ & (c_j(d_j,e_j) \rightarrow \alpha_j(d_j,e_j) \lhd t_j(d_j,e_j) \cdot Y(g_j(d_j,e_j)) + \\ & \neg c_j(d_j,e_j) \rightarrow \delta \lhd 0) \\ = \; & \{ \text{ Lemma A.3, T1, S1 } \} \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \rightarrow \\ & \alpha_i(d_i,e_i) \lhd t_i(d_i,e_i) \cdot X(g_i(d_i,e_i)) \mid \alpha_j(d_j,e_j) \lhd t_j(d_j,e_j) \cdot Y(g_j(d_j,e_j)) \\ = \; & \{ \text{ S6, S9, S1 } \} \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \rightarrow \\ & \alpha_i(d_i,e_i) \mid \alpha_j(d_j,e_j) \lhd t_i(d_i,e_i) \lhd t_j(d_j,e_j) \cdot (X(g_i(d_i,e_i)) \parallel Y(g_j(d_j,e_j))) \\ = \; & \{ \text{ TA1 } \} \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \rightarrow \\ & (t_i(d_i,e_i) = t_j(d_j,e_j)) \rightarrow \alpha_i(d_i,e_i) \mid \alpha_j(d_j,e_j) \lhd t_i(d_i,e_i) \cdot (X(g_i(d_i,e_i)) \parallel Y(g_j(d_j,e_j))) \\ & \diamond \delta \lhd min(t_i(d_i,e_i),t_j(d_j,e_j)) \\ = \; & \{ \text{ Cond, Lemma A.5 } \} \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) = t_j(d_j,e_j) \rightarrow \\ & \alpha_i(d_i,e_i) \mid \alpha_j(d_j,e_j) \lhd t_i(d_i,e_i) \cdot (X(g_i(d_i,e_i)) \parallel Y(g_j(d_j,e_j))) + \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) \neq t_j(d_j,e_j) \rightarrow \\ & \delta \lhd min(t_i(d_i,e_i),t_j(d_j,e_j)) \cdot (X(d_i) \parallel Y(d_j)) \end{aligned}$$

□

Theorem 11.2. For TLPE's X, Y and Z where $Z(d_i,d_j) = X(d_i) \parallel Y(d_j)$, it holds that:

$$\begin{aligned} Z(d_i,d_j) \quad = \quad & \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) < t_j(d_j,e_j) \rightarrow \\ & \alpha_i(d_i,e_i) \lhd t_i(d_i,e_i) \cdot Z(g_i(d_i,e_i),d_j) + \\ & \textstyle\sum_{i\in I_x,e_i:E_i,j\in J_x,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_j(d_j,e_j) < t_i(d_i,e_i) \rightarrow \\ & \alpha_j(d_j,e_j) \lhd t_j(d_j,e_j) \cdot Z(d_i,g_j(d_j,e_j)) + \\ & \textstyle\sum_{i\in I_x,e_i:E_i} \sum_{j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \wedge t_i(d_i,e_i) = t_j(d_j,e_j) \rightarrow \\ & \alpha_i(d_i,e_i) \mid \alpha_j(d_j,e_j) \lhd t_i(d_i,e_i) \cdot Z(g_i(d_i,e_i),g_j(d_j,e_j)) + \\ & \textstyle\sum_{i\in I_x,e_i:E_i} \sum_{j\in I_y,e_j:E_j} c_i(d_i,e_i) \wedge c_j(d_j,e_j) \rightarrow \\ & \delta \lhd min(t_i(d_i,e_i),t_j(d_j,e_j)) \cdot Z(d_i,d_j) \end{aligned}$$

Proof.

$$
\begin{aligned}
& Z(d_i, d_j) \\
= \{ & \text{ definition } Z \} \\
& X(d_i) \parallel Y(d_j) \\
= \{ & \text{ M } \} \\
& X(d_i) \lfloor\!\lfloor Y(d_j) + Y(d_j) \lfloor\!\lfloor X(d_i) + X(d_i) \mid Y(d_j) \\
= \{ & \text{ Lemma A.7 } \} \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) < t_j(d_j, e_j) \rightarrow \\
& \alpha_i(d_i, e_i) \triangleleft t_i(d_i, e_i) \cdot (X(g_i(d_i, e_i)) \parallel Y(d_j)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) \geq t_j(d_j, e_j) \rightarrow \\
& \delta \triangleleft min(t_i(d_i, e_i), t_j(d_j, e_j)) \cdot (X(d_i) \parallel Y(d_j)) + \\
& Y(d_j) \lfloor\!\lfloor X(d_i) + X(d_i) \mid Y(d_j) \\
= \{ & \text{ Lemma A.7 } \} \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) < t_j(d_j, e_j) \rightarrow \\
& \alpha_i(d_i, e_i) \triangleleft t_i(d_i, e_i) \cdot (X(g_i(d_i, e_i)) \parallel Y(d_j)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) \geq t_j(d_j, e_j) \rightarrow \\
& \delta \triangleleft min(t_i(d_i, e_i), t_j(d_j, e_j)) \cdot (X(d_i) \parallel Y(d_j)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_j(d_j, e_j) < t_i(d_i, e_i) \rightarrow \\
& \alpha_j(d_j, e_j) \triangleleft t_j(d_j, e_j) \cdot (Y(g_j(d_j, e_j)) \parallel X(d_i)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_j(d_j, e_j) \geq t_i(d_i, e_i) \rightarrow \\
& \delta \triangleleft min(t_j(d_j, e_j), t_i(d_i, e_i)) \cdot (Y(d_j) \parallel X(d_i)) + \\
& X(d_i) \mid Y(d_j) \\
= \{ & \text{ Lemma A.8 } \} \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) < t_j(d_j, e_j) \rightarrow \\
& \alpha_i(d_i, e_i) \triangleleft t_i(d_i, e_i) \cdot (X(g_i(d_i, e_i)) \parallel Y(d_j)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) \geq t_j(d_j, e_j) \rightarrow \\
& \delta \triangleleft min(t_i(d_i, e_i), t_j(d_j, e_j)) \cdot (X(d_i) \parallel Y(d_j)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_j(d_j, e_j) < t_i(d_i, e_i) \rightarrow \\
& \alpha_j(d_j, e_j) \triangleleft t_j(d_j, e_j) \cdot (Y(g_j(d_j, e_j)) \parallel X(d_i)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_j(d_j, e_j) \geq t_i(d_i, e_i) \rightarrow \\
& \delta \triangleleft min(t_j(d_j, e_j), t_i(d_i, e_i)) \cdot (Y(d_j) \parallel X(d_i)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i} \sum_{j \in I_y, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) = t_j(d_j, e_j) \rightarrow \\
& \alpha_i(d_i, e_i) \mid \alpha_j(d_j, e_j) \triangleleft t_i(d_i, e_i) \cdot (X(g_i(d_i, e_i)) \parallel Y(g_j(d_j, e_j))) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i} \sum_{j \in I_y, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) \neq t_j(d_j, e_j) \rightarrow \\
& \delta \triangleleft min(t_i(d_i, e_i), t_j(d_j, e_j)) \cdot (X(d_i) \parallel Y(d_j)) \\
= \{ & \text{ } t_i(d_i, e_j) \geq t_j(d_j, e_j) \vee t_j(d_j, e_j) \geq t_i(d_i, e_i) \vee t_i(d_i, e_i) \neq t_j(d_j, e_j) \equiv true, \text{ Lemma A.6 } \} \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) < t_j(d_j, e_j) \rightarrow \\
& \alpha_i(d_i, e_i) \triangleleft t_i(d_i, e_i) \cdot (X(g_i(d_i, e_i)) \parallel Y(d_j)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_j(d_j, e_j) < t_i(d_i, e_i) \rightarrow \\
& \alpha_j(d_j, e_j) \triangleleft t_j(d_j, e_j) \cdot (X(d_i) \parallel Y(g_j(d_j, e_j))) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i} \sum_{j \in I_y, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) = t_j(d_j, e_j) \rightarrow \\
& \alpha_i(d_i, e_i) \mid \alpha_j(d_j, e_j) \triangleleft t_i(d_i, e_i) \cdot (X(g_i(d_i, e_i)) \parallel Y(g_j(d_j, e_j))) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i} \sum_{j \in I_y, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \rightarrow \\
& \delta \triangleleft min(t_i(d_i, e_i), t_j(d_j, e_j)) \cdot (X(d_i) \parallel Y(d_j)) \\
= \{ & \text{ definition } Z \} \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) < t_j(d_j, e_j) \rightarrow \\
& \alpha_i(d_i, e_i) \triangleleft t_i(d_i, e_i) \cdot Z(g_i(d_i, e_i), d_j) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i, j \in J_x, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_j(d_j, e_j) < t_i(d_i, e_i) \rightarrow \\
& \alpha_j(d_j, e_j) \triangleleft t_j(d_j, e_j) \cdot Z(d_i, g_j(d_j, e_j)) + \\
& \textstyle\sum_{i \in I_x, e_i : E_i} \sum_{j \in I_y, e_j : E_j} c_i(d_i, e_i) \wedge c_j(d_j, e_j) \wedge t_i(d_i, e_i) = t_j(d_j, e_j) \rightarrow \\
& \alpha_i(d_i, e_i) \mid \alpha_j(d_j, e_j) \triangleleft t_i(d_i, e_i) \cdot Z(g_i(d_i, e_i), g_j(d_j, e_j)) +
\end{aligned}
$$

$\sum_{i\in I_x,e_i:E_i}\sum_{j\in I_y,e_j:E_j} c_i(d_i,e_i)\wedge c_j(d_j,e_j)\rightarrow$
$\delta \triangleleft min(t_i(d_i,e_i),t_j(d_j,e_j))\cdot Z(d_i,d_j)$

□

Lemma A.9. $\alpha \triangleleft t = \alpha \triangleleft t + \sum_{u:\mathbb{T}} u < t \rightarrow \delta \triangleleft u$

Proof.

$\alpha \triangleleft t$
$=$ { T3 }
$\sum_{u:\mathbb{T}} \alpha \triangleleft t \triangleleft u$
$=$ { TA1 }
$\sum_{u:\mathbb{T}} t = u \rightarrow \alpha \triangleleft t \diamond \delta \triangleleft min(t,u)$
$=$ { $\sum_{u:\mathbb{T}} t \neq u \rightarrow \delta \triangleleft min(t,u) \equiv \sum_{u:\mathbb{T}} t < u \rightarrow \delta \triangleleft u$ }
$\sum_{u:\mathbb{T}} t = u \rightarrow \alpha \triangleleft t \diamond u < t \rightarrow \delta \triangleleft u$
$=$ { SUM1 }
$\alpha \triangleleft t + \sum_{u:\mathbb{T}} u < t \rightarrow \delta \triangleleft u$

□

Theorem 11.4. For any TLPE X, there exists a DS-TLPE Y such that $X = Y$.

Proof. The theorem follows directly from Lemma A.9. □

Appendix B

Axioms for processes

MA1	$\alpha\|\beta = \beta\|\alpha$
MA2	$(\alpha\|\beta)\|\gamma = \alpha\|(\beta\|\gamma)$
MA3	$\alpha\|\tau = \alpha$
MD1	$\tau \setminus \alpha = \tau$
MD2	$\alpha \setminus \tau = \alpha$
MD3	$\alpha \setminus (\beta\|\gamma) = (\alpha \setminus \beta) \setminus \gamma$
MD4	$(a(d)\|\alpha) \setminus a(d) = \alpha$
MD5	$(a(d)\|\alpha) \setminus b(e) = a(d)\|(\alpha \setminus b(e))$ if $a \not\equiv b$ or $d \not\approx e$
MS1	$\tau \sqsubseteq \alpha = \mathit{true}$
MS2	$a \sqsubseteq \tau = \mathit{false}$
MS3	$a(d)\|\alpha \sqsubseteq a(d)\|\beta = \alpha \sqsubseteq \beta$
MS4	$a(d)\|\alpha \sqsubseteq b(e)\|\beta = a(d)\|(\alpha \setminus b(e)) \sqsubseteq \beta$ if $a \not\equiv b$ or $d \not\approx e$
MAN1	$\underline{\tau} = \tau$
MAN2	$\underline{a(d)} = a$
MAN3	$\underline{\alpha\|\beta} = \underline{\alpha}\|\underline{\beta}$
A1	$x + y = y + x$
A2	$x + (y + z) = (x + y) + z$
A3	$x + x = x$
A4	$(x + y)\cdot z = x\cdot z + y\cdot z$
A5	$(x\cdot y)\cdot z = x\cdot(y\cdot z)$
A6	$\alpha + \delta = \alpha$
A7	$\delta\cdot x = \delta$
Cond1	$\mathit{true} \rightarrow x \diamond y = x$
Cond2	$\mathit{false} \rightarrow x \diamond y = y$
SUM1	$\sum_{d:D} x = x$
SUM3	$\sum_{d:D} X(d) = \sum_{d:D} X(d) + X(d)$
SUM4	$\sum_{d:D}(X(d) + Y(d)) = \sum_{d:D} X(d) + \sum_{d:D} Y(d)$
SUM5	$(\sum_{d:D} X(d))\cdot y = \sum_{d:D} X(d)\cdot y$

M	$x \parallel y = x \between y + y \between x + x \mid y$
LM1	$\alpha \between x = (\alpha \ll x) \cdot x$
LM2	$\delta \between x = \delta \ll x$
LM3	$\alpha \cdot x \between y = (\alpha \ll y) \cdot (x \parallel y)$
LM4	$(x + y) \between z = x \between z + y \between z$
LM5	$(\sum_{d:D} X(d)) \between y = \sum_{d:D} X(d) \between y$
LM6	$x \triangleleft t \between y = (x \between y) \triangleleft t$
S1	$x \mid y = y \mid x$
S2	$(x \mid y) \mid z = x \mid (y \mid z)$
S3	$x \mid \tau = x$
S4	$\alpha \mid \delta = \delta$
S5	$(\alpha \cdot x) \mid \beta = \alpha \mid \beta \cdot x$
S6	$(\alpha \cdot x) \mid (\beta \cdot y) = \alpha \mid \beta \cdot (x \parallel y)$
S7	$(x + y) \mid z = x \mid z + y \mid z$
S8	$(\sum_{d:D} X(d)) \mid y = \sum_{d:D} X(d) \mid y$
S9	$x \triangleleft t \mid y = (x \mid y) \triangleleft t$
TB1	$x \ll \alpha = x$
TB2	$x \ll \delta = x$
TB3	$x \ll y \triangleleft t = \sum_{u:\mathbb{T}} u < t \to (x \triangleleft u) \ll y$
TB4	$x \ll (y + z) = x \ll y + x \ll z$
TB5	$x \ll y \cdot z = x \ll y$
TB6	$x \ll \sum_{d:D} Y(d) = \sum_{d:D} x \ll Y(d)$
TC1	$(x \between y) \between z = x \between (y \parallel z)$
TC2	$x \between \delta = x \cdot \delta$
TC3	$(x \mid y) \between z = x \mid (y \between z)$
R1	$\rho_R(\tau) = \tau$
R2	$\rho_R(a(d)) = b(d)$ if $a \to b \in R$ for some b
R3	$\rho_R(a(d)) = a(d)$ if $a \to b \notin R$ for all b
R4	$\rho_R(\alpha \mid \beta) = \rho_R(\alpha) \mid \rho_R(\beta)$
R5	$\rho_R(\delta) = \delta$
R6	$\rho_R(x + y) = \rho_R(x) + \rho_R(y)$
R7	$\rho_R(x \cdot y) = \rho_R(x) \cdot \rho_R(y)$
R8	$\rho_R(\sum_{d:D} X(d)) = \sum_{d:D} \rho_R(X(d))$
R9	$\rho_R(x \triangleleft t) = \rho_R(x) \triangleleft t$

T1	$x + \delta \triangleleft 0 = x$
T2	$c \to x = c \to x \diamond \delta \triangleleft 0$
T3	$x = \sum_{t:\mathbb{T}} x \triangleleft t$
T4	$x \triangleleft t \cdot y = x \triangleleft t \cdot (t \gg y)$
T5	$x \triangleleft t = t > 0 \to x \triangleleft t$
TA1	$\alpha \triangleleft t \triangleleft u = (t \approx u) \to \alpha \triangleleft t \diamond \delta \triangleleft \min(t, u)$
TA2	$\delta \triangleleft t \triangleleft u = \delta \triangleleft \min(t, u)$
TA3	$(x + y) \triangleleft t = x \triangleleft t + y \triangleleft t$
TA4	$(x \cdot y) \triangleleft t = x \triangleleft t \cdot y$
TA5	$(\sum_{d:D} X(d)) \triangleleft t = \sum_{d:D} X(d) \triangleleft t$
TI1	$t \gg \alpha \triangleleft u = t < u \to \alpha \triangleleft u \diamond \delta \triangleleft t$
TI2	$t \gg \delta \triangleleft u = \delta \triangleleft \max(t, u)$
TI3	$t \gg (x + y) = t \gg x + t \gg y$
TI4	$t \gg (x \cdot y) = (t \gg x) \cdot y$
TI5	$t \gg \sum_{d:D} X(d) = \sum_{d:D} t \gg X(d)$

C1	$\Gamma_C(\alpha) = \gamma_C(\alpha)$	C4	$\Gamma_C(x{\cdot}y) = \Gamma_C(x){\cdot}\Gamma_C(y)$
C2	$\Gamma_C(\delta) = \delta$	C5	$\Gamma_C(\sum_{d:D} X(d)) = \sum_{d:D} \Gamma_C(X(d))$
C3	$\Gamma_C(x+y) = \Gamma_C(x) + \Gamma_C(y)$	C6	$\Gamma_C(x \triangleleft t) = \Gamma_C(x) \triangleleft t$
V1	$\nabla_V(\alpha) = \alpha$ if $\underline{\alpha} \subseteq V \cup \{\tau\}$	V4	$\nabla_V(x+y) = \nabla_V(x) + \nabla_V(y)$
V2	$\nabla_V(\alpha) = \delta$ if $\underline{\alpha} \not\subseteq V \cup \{\tau\}$	V5	$\nabla_V(x{\cdot}y) = \nabla_V(x){\cdot}\nabla_V(y)$
V3	$\nabla_V(\delta) = \delta$	V6	$\nabla_V(\sum_{d:D} X(d)) = \sum_{d:D} \nabla_V(X(d))$
		V7	$\nabla_V(x \triangleleft t) = \nabla_V(x) \triangleleft t$
TV1	$\nabla_V(\nabla_W(x)) = \nabla_{V \cap W}(x)$		
E1	$\partial_B(\tau) = \tau$	E5	$\partial_B(\delta) = \delta$
E2	$\partial_B(a(d)) = a(d)$ if $a \notin B$	E6	$\partial_B(x+y) = \partial_B(x) + \partial_B(y)$
E3	$\partial_B(a(d)) = \delta$ if $a \in B$	E7	$\partial_B(x{\cdot}y) = \partial_B(x){\cdot}\partial_B(y)$
E4	$\partial_B(\alpha \mid \beta) = \partial_B(\alpha) \mid \partial_B(\beta)$	E8	$\partial_B(\sum_{d:D} X(d)) = \sum_{d:D} \partial_B(X(d))$
		E9	$\partial_B(x \triangleleft t) = \partial_B(x) \triangleleft t$
H1	$\tau_I(\tau) = \tau$	H5	$\tau_I(\delta) = \delta$
H2	$\tau_I(a(d)) = \tau$ if $a \in I$	H6	$\tau_I(x{+}y) = \tau_I(x) + \tau_I(y)$
H3	$\tau_I(a(d)) = a(d)$ if $a \notin I$	H7	$\tau_I(x{\cdot}y) = \tau_I(x){\cdot}\tau_I(y)$
H4	$\tau_I(\alpha \mid \beta) = \tau_I(\alpha) \mid \tau_I(\beta)$	H8	$\tau_I(\sum_{d:D} X(d)) = \sum_{d:D} \tau_I(X(d))$
		H9	$\tau_I(x \triangleleft t) = \tau_I(x) \triangleleft t$
U1	$\Upsilon_U(\tau) = \tau$	U5	$\Upsilon_U(\delta) = \delta$
U2	$\Upsilon_U(a(d)) = int$ if $a \in U$	U6	$\Upsilon_U(x{+}y) = \Upsilon_U(x) + \Upsilon_U(y)$
U3	$\Upsilon_U(a(d)) = a(d)$ if $a \notin U$	U7	$\Upsilon_U(x{\cdot}y) = \Upsilon_U(x){\cdot}\Upsilon_U(y)$
U4	$\Upsilon_U(\alpha \mid \beta) = \Upsilon_U(\alpha) \mid \Upsilon_U(\beta)$	U8	$\Upsilon_U(\sum_{d:D} X(d)) = \sum_{d:D} \Upsilon_U(X(d))$
		U9	$\Upsilon_U(x \triangleleft t) = \Upsilon_U(x) \triangleleft t$

B1	$x{\cdot}\tau = x$
B2	$x{\cdot}(\tau{\cdot}(y+z)+y) = x{\cdot}(y+z)$
W1	$x{\cdot}\tau = x$
W2	$\tau{\cdot}x = \tau{\cdot}x + x$
W2	$a{\cdot}(\tau{\cdot}x+y) = a{\cdot}(\tau{\cdot}x+y) + a{\cdot}x$

Failures equivalence	$a{\cdot}(b{\cdot}x+u) + a{\cdot}(b{\cdot}y+v) = a{\cdot}(b{\cdot}x+b{\cdot}y+u) + a{\cdot}(b{\cdot}x+b{\cdot}y+v)$
	$a{\cdot}x + a{\cdot}(y+z) = a{\cdot}x + a{\cdot}(x+y) + a{\cdot}(y+z)$
Trace equivalence	$x{\cdot}(y+z) = x{\cdot}y + x{\cdot}z$
Weak trace equivalence	$x{\cdot}(y+z) = x{\cdot}y + x{\cdot}z$
	$\tau{\cdot}x = x$
	$x{\cdot}\tau = x$

Appendix C

Complete mCRL2 source file

In this appendix, the entire mCRL2 source file as used in the analysis of the complete system in Chapter 10 is described. Subsequent alterations on this model are discussed in the respective section numbers of Chapter 10.

system.mcrl2

```
sort Direction = Nat;
     Coord = struct coord(x:Int,y:Int);

map  DIM: Pos;
     LEADERNUM: Nat;
eqn  DIM = 2;
     LEADERNUM = 0;

sort RList = List(Nat);
map  RotationList: RList;
eqn  RotationList = [ 0, 1, 2, 3 ];

map  mapNodeId : Nat -> Nat;
var  n:Nat;
eqn  mapNodeId(n) = (n - LEADERNUM) mod DIM*DIM;

map  makeMinCoords, makeMaxCoords: Coord;
var  a, b: Int;
eqn  makeMinCoords = coord(-a, -((DIM-1)-b))
     whr a = (LEADERNUM mod DIM),
         b = (LEADERNUM div DIM) end;
     makeMaxCoords = coord((DIM-1)-a, b)
     whr a = (LEADERNUM mod DIM),
         b = (LEADERNUM div DIM) end;

map  maxCoord : Coord#Coord -> Coord;
     minCoord : Coord#Coord -> Coord;
var  x1,y1,x2,y2: Int;
eqn  maxCoord(coord(x1,y1),coord(x2,y2)) = coord(max(x1,x2),max(y1,y2));
     minCoord(coord(x1,y1),coord(x2,y2)) = coord(min(x1,x2),min(y1,y2));

map  move: Direction#Coord->Coord;
var  x,y: Int;
eqn  move(0,coord(x,y)) = coord(x  ,y+1);
     move(1,coord(x,y)) = coord(x+1,y);
     move(2,coord(x,y)) = coord(x  ,y-1);
     move(3,coord(x,y)) = coord(x-1,y);
```

```
38
39  map  calcCoord : Int -> Coord;
40  var  num:Int;
41  eqn  calcCoord(num) =
42               coord(num mod DIM - LEADERNUM mod DIM,
43                     LEADERNUM div DIM - num div DIM);
44
45  map  rr: Direction#Int -> Nat;
46  var  d:Direction;
47       n:Int;
48  eqn  rr(d,n) = (n+d) mod 4;
49  map  rd: Direction -> Direction;
50  var  d: Direction;
51  eqn  rd(d) = (d + 2) mod 4;
52
53  map  rl: Direction#Int -> Nat;
54  var  d: Direction;
55       e: Int;
56  eqn  rl(d,e) = (d-e) mod 4;
57
58  map  neighbornum: Int#Direction->Int;
59  var  n:Int;
60  eqn  neighbornum(n,0) = if(n div DIM>0,n-DIM,n);
61       neighbornum(n,1) = if(n mod DIM<DIM-1,n+1,n);
62       neighbornum(n,2) = if(n div DIM<DIM-1,n+DIM,n);
63       neighbornum(n,3) = if(n mod DIM>0,n-1,n);
64
65  map  diffCoord: Coord#Coord -> Coord;
66  var  x1,y1,x2,y2:Int;
67  eqn  diffCoord(coord(x1,y1),coord(x2,y2)) = coord(x1-x2,y2-y1);
68
69  act  % phase 1
70       becomeLeader: Int;
71       gotLeader,
72       monLeader,
73       commLeader: Int#Int#Bool;       % num,leaderid,newleader
74       sendID,
75       recvID,
76       commID: Int#Int;                % num,id
77
78       % phase 2
79       sendCoord,
80       recvCoord,
81       commCoord: Int#Coord#Direction#Direction;
82       reportCoord,
83       ackCoord,
84       coordAction: Int#Nat#Coord;
85
86       % phase 3
87       sendMinMax,
88       recvMinMax,
89       commMinMax: Int#Int#Coord#Coord; % num,lid,min,max
90       reportMinMax,
91       checkMinMax,
92       commMinMax2: Int#Coord#Coord; % num,min,max
93
94       % phase 4
95       sendUpdate,
96       recvUpdate,
97       commUpdate: Int#Int#Nat; % num,leaderid,state
98       updateLight: Int#Nat; % num,newstate
99       timer;
100
101      % phase transitions
102      phtry, phack, phcomm: Nat;
103      intern,
104      error;
```

```
105       reset;
106
107       % reset actions
108       sendReset,
109       recvReset,
110       commReset: Int#Nat;
111       sendMonReset,
112       recvMonReset,
113       commMonReset;
114
115   proc Smartpixellaunch(num:Nat) =
116        Smartpixelinit(num,0);
117       Smartpixelinit(num,nrc:Nat) =
118        Smartpixel(num,mapNodeId(num),1,0,4,4,4,4,false,coord(0,0),coord(0,0),coord(0,0),
119         0,0,nrc);
120       Smartpixelreset(num,nrc:Nat) =
121        Smartpixel(num,mapNodeId(num),0,0,4,4,4,0,false,coord(0,0),coord(0,0),coord(0,0),
122         0,0,nrc);
123
124       Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr:Nat,leader:Bool,log,mincd,maxcd:Coord,
125         f,r,rc:Nat) =
126
127       %%%% PHASE 1: LEADER ELECTION %%%%
128
129       % if we haven't sent our ID in a given direction, do so
130       (ph==1&&sl<4&&(neighbornum(num,rr(sl,RotationList.num))==num)) ->
131              intern .
132              Smartpixel(num,leader_id,ph,sl+1,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
133       (ph==1&&sl<4&&(neighbornum(num,rr(sl,RotationList.num))!=num)) ->
134              sendID(neighbornum(num,rr(sl,RotationList.num)),leader_id) .
135              Smartpixel(num,leader_id,ph,sl+1,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
136
137       % if we receive a lower ID than our leader ID, honor it - and retransmit it to the
138       % connected nodes
139       sum rid:Nat . (ph==1&&rid<leader_id) -> recvID(num,rid) .
140              gotLeader(num,rid,mapNodeId(num)==leader_id) .
141              Smartpixel(num,rid,ph,0,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
142       % if we receiver our own ID or a higher ID than our leader id, do nothing
143       sum rid:Nat . (ph==1&&rid>=leader_id) ->  recvID(num,rid) .
144              Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
145       % we are the leader - make the initial transmission
146       (ph==1&&sl==4&&mapNodeId(num)==leader_id) -> phtry(1) . becomeLeader(num) .
147              Smartpixel(num,leader_id,2,sl,0,ss,su,sr,true,log,mincd,maxcd,f,r,rc) +
148       % we are not the leader - do not send
149       (ph==1&&sl==4&&mapNodeId(num)!=leader_id) -> phtry(1) .
150              Smartpixel(num,leader_id,2,sl,sc,ss,su,sr,false,log,mincd,maxcd,f,r,rc) +
151
152       %%%% PHASE 2: COORDINATE DETERMINATION %%%%
153
154       % if we haven't sent position information in a given direction, do so
155       (ph==2&&sc<4&&neighbornum(num,rr(sc,RotationList.num))==num) -> intern .
156              Smartpixel(num,leader_id,ph,sl,sc+1,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
157       sum d':Direction.(ph==2&&sc<4&&neighbornum(num,rr(sc,RotationList.num))!=num) ->
158              sendCoord(neighbornum(num,rr(sc,RotationList.num)),move(rr(sc,r),log),
159                        rr(rd(sc),r),d') .
160              Smartpixel(num,leader_id,ph,sl,sc+1,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
161
162       % if we receive a new logical coordinates, believe them and update accordingly
163       % - but only if we didn't know our position in the first place
164       sum c:Coord,d,d':Direction . (ph==2&&log==coord(0,0)&&!leader&&
165                                     d'==rl(d,RotationList.num)) ->
166              recvCoord(num,c,d,d') . reportCoord(num,rl(d,d'),c) .
167              Smartpixel(num,leader_id,ph,sl,0,ss,su,sr,leader,c,c,c,f,rl(d,d'),rc) +
168
169       % if we receive any new position which conflicts with what we already have,
170       % initiate a reset action
171       sum c:Coord,d,d':Direction . (ph==2&&c!=log&&(log!=coord(0,0)||leader)&&
```

```
                                0<=d'&&d'<=3) ->
       recvCoord(num,c,d,d') . sendMonReset . Smartpixelreset(num,rc) +

% if we receive a position and the coordinates/direction is ok, ignore it
sum d,d':Direction . (ph==2&&(log!=coord(0,0)||leader)&&
                      d'==rl(d,RotationList.num)&&r==rl(d,d')) ->
       recvCoord(num,log,d,d') .
       Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +

% if we receive a position and the coordinates are ok but the direction is not, reset
sum d,d':Direction . (ph==2&&(log!=coord(0,0)||leader)&&d'==rl(d,RotationList.num)&&
                      r!=rl(d,d')) ->
       recvCoord(num,log,d,d') . sendMonReset . Smartpixelreset(num,rc) +

(ph==2&&sc==4) -> phtry(2) .
       Smartpixel(num,leader_id,3,sl,sc,if(leader,0,ss),su,sr,leader,log,log,log,f,
                  r,rc) +

%%%% PHASE 3: GRID SIZE DETERMINATION %%%%
% if we haven't sent min/max information in a given direction, do so
(ph==3&&ss<4&&neighbornum(num,rr(ss,RotationList.num))==num) -> intern .
       Smartpixel(num,leader_id,ph,sl,sc,ss+1,su,sr,leader,log,mincd,maxcd,f,r,rc) +
(ph==3&&ss<4&&neighbornum(num,rr(ss,RotationList.num))!=num) ->
       sendMinMax(neighbornum(num,rr(ss,RotationList.num)),leader_id,mincd,maxcd) .
       Smartpixel(num,leader_id,ph,sl,sc,ss+1,su,sr,leader,log,mincd,maxcd,f,r,rc) +

% if we receive a new min/max coordinates from a higher leader id, this is
% wrong, so initiate a reset action
sum n:Nat,cmin,cmax:Coord . (ph==3&&n>leader_id) -> recvMinMax(num,n,cmin,cmax) .
        sendMonReset . Smartpixelreset(num,rc) +

% if we receive new min/max coordinates, believe them
sum cmin,cmax:Coord,lid:Nat . (ph==3&&lid<=leader_id&&
              (minCoord(cmin,mincd)!=mincd)||(maxCoord(cmax,maxcd)!=maxcd)) ->
       recvMinMax(num,lid,cmin,cmax) .
       reportMinMax(num,minCoord(cmin,mincd),maxCoord(cmax,maxcd)) .
       Smartpixel(num,lid,ph,sl,sc,0,su,sr,leader,log,
                minCoord(cmin,mincd),maxCoord(cmax,maxcd),f,r,rc) +

% if we receive uninteresting new min/max coordinates, ignore them
% but update our leader id
sum cmin,cmax:Coord,lid:Nat . (ph==3&&lid<=leader_id&&
           (minCoord(cmin,mincd)==mincd)&&(maxCoord(cmax,maxcd)==maxcd)) ->
       recvMinMax(num,lid,cmin,cmax) .
       Smartpixel(num,lid,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +

(ph==3&&ss==4) -> phtry(3) . updateLight(num,0) .
       Smartpixel(num,leader_id,4,sl,sc,ss,su,sr,leader,log,log,log,f,r,rc) +

%%%% PHASE 4: SHOWTIME %%%%
(ph==4&&su<4&&neighbornum(num,rr(su,RotationList.num))==num) -> intern .
       Smartpixel(num,leader_id,ph,sl,sc,ss,su+1,sr,leader,log,mincd,maxcd,f,r,rc) +
(ph==4&&su<4&&neighbornum(num,rr(su,RotationList.num))!=num) ->
       sendUpdate(neighbornum(num,rr(su,RotationList.num)),
                  if(leader,mapNodeId(num),leader_id),f) .
       Smartpixel(num,leader_id,ph,sl,sc,ss,su+1,sr,leader,log,mincd,maxcd,f,r,rc) +

% if we receive a new min/max coordinates from a higher different leader id, this
% is wrong, so initiate a reset action
sum a:Int,b:Nat . (ph==4&&a>leader_id) -> recvUpdate(num,a,b) . sendMonReset .
       Smartpixelreset(num,rc) +

% if we have an update, honor it
sum n,a:Nat . (ph==4&&n<=leader_id&&a!=f) -> recvUpdate(num,n,a) .
                    updateLight(num,a) .
                    Smartpixel(num,n,ph,sl,sc,ss,0,sr,leader,log,mincd,maxcd,a,r,rc) +
```

```
239       % if we have an update we already know, ignore it - but honor a possibly lower
240       % leader id
241       sum n:Nat.(ph==4&&n<=leader_id) -> recvUpdate(num,n,f) .
242              Smartpixel(num,n,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
243
244       % if we act as leader, initiate periodic updates
245       (ph==4&&leader) -> timer .
246              updateLight(num,(f+1) mod 2) .
247              Smartpixel(num,leader_id,ph,sl,sc,ss,0,sr,leader,log,mincd,maxcd,
248                         (f+1) mod 2,r,rc) +
249
250       % ignore out of phase-messages
251       sum n:Int. (ph!=1) -> recvID(num,n) .
252              Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
253       sum c:Coord,d,d':Direction . (ph!=2&&0<=d'&&d'<=3) -> recvCoord(num,c,d,d') .
254              Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
255       sum n:Int,c,d:Coord.(ph!=3) -> recvMinMax(num,n,c,d) .
256              Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
257       sum n:Int,a:Nat.(ph!=4) -> recvUpdate(num,n,a) .
258              Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
259
260       % if reset, reset!
261       sum x:Nat.(ph!=0&&x<rc) -> recvReset(num,x) .
262              Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
263       sum x:Nat.(ph!=0&&x>=rc) -> recvReset(num,x) .
264              Smartpixelreset(num,x) +
265       sum x:Nat.(ph==0) -> recvReset(num,x) .
266              Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr,leader,log,mincd,maxcd,f,r,rc) +
267
268       (ph==0&&sr<4&&(neighbornum(num,rr(sr,RotationList.num))==num)) -> intern .
269              Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr+1,leader,log,mincd,maxcd,f,r,rc) +
270       (ph==0&&sr<4&&(neighbornum(num,rr(sr,RotationList.num))!=num)) ->
271               sendReset(neighbornum(num,rr(sr,RotationList.num)),rc) .
272               Smartpixel(num,leader_id,ph,sl,sc,ss,su,sr+1,leader,log,mincd,maxcd,f,r,rc) +
273       (ph==0&&sr==4) -> intern . Smartpixelinit(num,(rc+1) mod 10);
274
275       MonitorInit =
276               Monitor(DIM*DIM-1,DIM*DIM-1,DIM*DIM);
277
278       Monitor(n,m,o:Int) =
279        % phase 1
280        sum a,b:Int.monLeader(a,b,true).Monitor(n-1,m,o) +
281        sum a,b:Int.monLeader(a,b,false).Monitor(n,m,o) +
282        % phase 2
283        sum num:Nat.ackCoord(num,RotationList.num,calcCoord(num)) . Monitor(n,m-1,o) +
284        sum num,r:Nat,c:Coord. (c!=calcCoord(num)||r!=RotationList.num) ->
285              ackCoord(num,r,c) . error +
286        % phase 3
287        sum id:Nat,cmin,cmax:Coord . ((x(cmax)-x(cmin)+1 == DIM) &&
288                                (y(cmax)-y(cmin)+1 == DIM) &&
289                                ((cmin!=makeMinCoords)||(cmax!=makeMaxCoords))) ->
290                               checkMinMax(id,cmin,cmax) . error +
291        sum id:Nat,cmin,cmax:Coord . ((x(cmax)-x(cmin)+1 == DIM) &&
292                                (y(cmax)-y(cmin)+1 == DIM) &&
293                                (cmin==makeMinCoords&&cmax==makeMaxCoords)) ->
294                               checkMinMax(id,cmin,cmax) . Monitor(n,m,o-1) +
295        sum id:Nat,cmin,cmax:Coord . ((x(cmax)-x(cmin)+1 <  DIM) ||
296                                (y(cmax)-y(cmin)+1 <  DIM)) ->
297                               checkMinMax(id,cmin,cmax) . Monitor(n,m,o) +
298        sum id:Nat,cmin,cmax:Coord . ((x(cmax)-x(cmin)+1 >  DIM) ||
299                                (y(cmax)-y(cmin)+1 >  DIM)) ->
300                               checkMinMax(id,cmin,cmax) . error +
301        (n<=0) -> phack(1).Monitor(n,m,o) +
302        (m<=0) -> phack(2).Monitor(n,m,o) +
303        (o<=0) -> phack(3).Monitor(n,m,o) +
304        recvMonReset . MonitorInit;
305
```

```
306 proc    System =
307      hide({becomeLeader}, (
308       allow({commID,commLeader,commCoord,coordAction,commMinMax,commMinMax2,commUpdate,
309              updateLight,phcomm,intern,error,reset,becomeLeader,timer,commReset,
310              commMonReset}, (
311        comm({sendID|recvID->commID,gotLeader|monLeader->commLeader,
312              sendCoord|recvCoord->commCoord,reportCoord|ackCoord->coordAction,
313              sendMinMax|recvMinMax->commMinMax,reportMinMax|checkMinMax->commMinMax2,
314              sendUpdate|recvUpdate->commUpdate,
315              phtry|phack->phcomm,
316              sendReset|recvReset->commReset,
317              sendMonReset|recvMonReset->commMonReset}, (
318         Smartpixellaunch( 0) || Smartpixellaunch( 1) || Smartpixellaunch( 2) ||
319         Smartpixellaunch( 3) ||
320         MonitorInit
321        ))
322       ))
323      ));
324
325 init    System;
```

system.ren

```
1  act ctau;
2  var i1,i2:Int;
3      c1,c2:Coord;
4      d1,d2:Nat;
5      n1:Nat;
6      b1:Bool;
7  rename
8    % phase 1
9    commID(i1,i2) => tau;
10   commLeader(i1,i2,b1) => ctau;
11   % phase 2
12   commCoord(i1,c1,d1,d2) => tau;
13   coordAction(i1,n1,c1) => ctau;
14   % phase 3
15   commMinMax(i1,i2,c1,c2) => tau;
16   commMinMax2(i1,c1,c2) => ctau;
17   % phase 4
18   commUpdate(i1,i2,n1) => tau;
19   updateLight(i1,n1) => ctau;
20   phcomm(n1) => ctau;
21   intern => ctau;
22   commReset(i1,n1) => ctau;
```

Bibliography

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for communication and mutual exclusion. In *Report CS-R8409*. Centrum voor Wiskunde & Informatica, Amsterdam, 1984.

[BLL+95] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[CR79] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22:281–283, 1979.

[GMR+07] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. <http://drops.dagstuhl.de/opus/volltexte/2007/862> [date of citation: 2007-01-01].

[GR07] J. F. Groote and M. A. Reniers. *Designing and understanding the behaviour of systems: Lecture notes for RADV 2IW25 (2007/2008)*. Department of Computer Science, Eindhoven University of Technology, Eindhoven, 2007.

[Hen08] W. Hendriksen. Smartpixel 2008. Unpublished, 2008.

[HR04] M. Huth and M. Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2nd edition, 2004.

[IR90] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88:60–87, 1990.

[Lan77] G. Le Lann. Distributed systems - towards a formal approach. *Information Processing*, 77:155–160, 1977.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[RGvdZvW02] M. A. Reniers, J. F. Groote, M. B. van der Zwaag, and J. van Wamel. Completeness of timed μCRL. *Fundamenta Informaticae*, 50:1–42, 2002.

[Use02] Y. S. Usenko. *Lineralization in μCRL*. PhD thesis, Technische Universiteit Eindhoven, 2002.